My ZIP isn't your ZIP: Identifying and Exploiting Semantic Gaps Between ZIP Parsers

Yufan You[1], Jianjun Chen[1,2,✉], Qi Wang[1], and Haixin Duan[1,2]

[1]Tsinghua University
[2]Zhongguancun Laboratory

Abstract

ZIP is one of the most popular archive formats. It is used not only as archive files, but also as the container for other file formats, including office documents, Android applications, Java archives, and many more. Despite its ubiquity, the ZIP file format specification is imprecisely specified, posing the risk of semantic gaps between implementations that can be exploited by attackers. While prior research has reported individual such vulnerabilities, there is a lack of systematic studies for ZIP parsing ambiguities.

In this paper, we developed a differential fuzzer ZIPDIFF and systematically identified parsing inconsistencies between 50 ZIP parsers across 19 programming languages. The evaluation results show that almost all pairs of parsers are vulnerable to certain parsing ambiguities. We summarize our findings as 14 distinct parsing ambiguity types in three categories with detailed analysis, systematizing current knowledge and uncovering 10 types of new parsing ambiguities. We demonstrate five real-world scenarios where these parsing ambiguities can be exploited, including bypassing secure email gateways, spoofing office document content, impersonating VS Code extensions, and tampering with signed nested JAR files while still passing Spring Boot's signature verification. We further propose seven mitigation strategies to address these ambiguities. We responsibly reported the vulnerabilities to the affected vendors and received positive feedback, including bounty rewards from Gmail, Coremail, and Zoho, and three CVEs from Go, LibreOffice, and Spring Boot.

1 Introduction

The ZIP file format is one of the most popular archive formats. Most users recognize ZIP by its `.zip` file extension, but they are also using it implicitly, when they are reading and writing office documents, installing Android applications and browser extensions, or running Java applications, because all these file formats use ZIP as the underlying container.

✉Corresponding author: jianjun@tsinghua.edu.cn

As ZIP has become a fundamental building block in many applications, security researchers have been investigating the file format to find potential vulnerabilities. Path traversal in ZIP filenames, a.k.a. *ZIP Slip* [38], results in arbitrary file writing that may escalate to remote code execution, while *ZIP bomb* can be used to conduct DoS attacks by exhausting computing resources with highly compressed ZIP archives. Many ZIP libraries are deploying fuzz testing to uncover memory bugs, including libzip, minizip, zip-rs, zip4j, zt-zip, and Commons Compress in OSS-Fuzz [6].

The work mentioned above all focused on individual ZIP parsers, while increasing attention from researchers is being directed toward semantic gaps—subtle inconsistencies between various ZIP implementations that can be exploited by attackers. A high-profile example is the infamous Android master key vulnerability [20], which bypassed Android's security by exploiting a mismatch between the ZIP component that verifies signatures for privileged applications and the component that decompresses the file contents. This discrepancy allowed malicious code to be inserted into privileged applications without breaking their signatures. While individual vulnerabilities have been discovered [16, 20, 23], these studies still rely on manual and ad hoc methods for discovery. To date, no systematic study has yet been conducted for ZIP parsing ambiguities.

In this paper, we developed a differential fuzzing tool ZIPDIFF. It generates and mutates ZIP files with grammar-based rules and uses feedback from ZIP parsers to guide the mutation. We collected 50 ZIP parsers across 19 programming languages and used ZIPDIFF to identify parsing inconsistencies between them. The results can be divided into two parts. First, we showed that parsing inconsistencies are quite prevalent among ZIP parsers nowadays, as almost any pair of parsers is inconsistent in parsing ZIP files. Second, we investigated the ZIP files that caused the discrepancies and classified these parsing ambiguities as 14 distinct types in three categories: redundant metadata, file path processing, and ZIP structure positioning. Many of these parsing ambiguities were previously unknown, and we also uncovered new variants of known ambiguities and new techniques to bypass detection of ambiguous ZIP files.

We demonstrate the practical impact of those parsing ambiguities through five real-world exploitation scenarios: secure email gateway bypass, office document content spoofing and signature forgery, nested JAR signature forgery, and VS Code extension impersonation. These vulnerabilities affect a wide range of critical applications, including Gmail, Golang, Spring Boot, and LibreOffice. We propose seven mitigation strategies to defend against those identified issues. We have reported our findings to the affected vendors and coordinated in addressing these issues. We received positive feedback including bounty rewards from Gmail, Coremail, and Zoho, and three CVEs from Go, LibreOffice, and Spring Boot.

In summary, we make the following contributions:

- We designed and implemented our differential fuzzing tool ZipDiff[1] to systematically identify inconsistencies between ZIP parsers (Section 4).
- We demonstrated the prevalence of discrepancies among ZIP parsers through the evaluation of ZipDiff on 50 parsers across 19 programming languages. We discovered 14 distinct types of ZIP parsing ambiguities, of which 10 types are newly discovered (Section 5.2).
- We proved the broad real-world impact of these discrepancies by observing five attack scenarios where inconsistencies between ZIP parsers can be exploited (Section 6). We have responsibly reported the identfied vulnerabilities to the affected vendors (Section 7.1).
- We proposed countermeasures to mitigate these attacks in various situations (Section 7.2).

2 Background

2.1 The ZIP File Format

The ZIP file format, originally created by PKWARE in 1989, is a widely used archive format specified in a document called APPNOTE.txt [29]. Known for its popularity as an archive format with the `.zip` file extension, ZIP also serves as a container for various other file types. These include office document formats OOXML and ODF, Java Archive (JAR), PHP Archive (PHAR), Visual Studio Extension (VSIX), Android Package (APK), Cross-Platform Install (XPI) used by Mozilla Firefox, Chrome Extension (CRX), and many more [15].

A regular ZIP file includes three major parts, the local file entries, the central directory, and the end of central directory record, as illustrated in Fig. 1. Each of the local file entries contains a Local File Header (LFH) and the compressed file data, optionally with an encryption header and a data descriptor. The central directory consists of the Central Directory Headers (CDHs), each pointing to an LFH. The CDHs and the LFHs provide the metadata for the file entries, such as filename, compression method, compressed and uncompressed sizes, CRC32 checksum, modification time, and several other flags. Many of these fields are redundant as they are stored twice in both the CDH and the LFH. The central directory reduces file I/O operations by keeping information spatially compact, while redundant metadata in the LFH can facilitate data recovery. The End Of Central Directory Record (EOCDR) provides information to locate the central directory and an optional file comment field.

[1] https://github.com/ouuan/ZipDiff

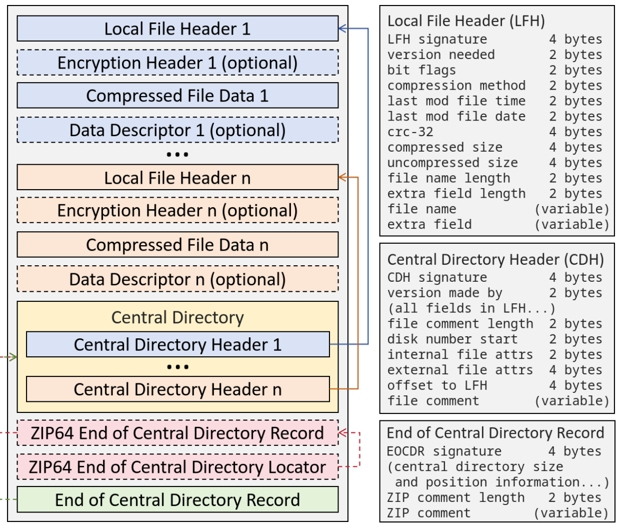
Local File Header 1
Encryption Header 1 (optional)
Compressed File Data 1
Data Descriptor 1 (optional)
Local File Header n
Encryption Header n (optional)
Compressed File Data n
Data Descriptor n (optional)
Central Directory
Central Directory Header 1
Central Directory Header n
ZIP64 End of Central Directory Record
ZIP64 End of Central Directory Locator
End of Central Directory Record
Local File Header (LFH)
LFH signature 4 bytes
version needed 2 bytes
bit flags 2 bytes
compression method 2 bytes
last mod file time 2 bytes
last mod file date 2 bytes
crc-32 4 bytes
compressed size 4 bytes
uncompressed size 4 bytes
file name length 2 bytes
extra field length 2 bytes
file name (variable)
extra field (variable)
Central Directory Header (CDH)
CDH signature 4 bytes
version made by 2 bytes
(all fields in LFH...)
file comment length 2 bytes
disk number start 2 bytes
internal file attrs 2 bytes
external file attrs 4 bytes
offset to LFH 4 bytes
file comment (variable)
End of Central Directory Record
EOCDR signature 4 bytes
(central directory size
and position information...)
ZIP comment length 2 bytes
ZIP comment (variable)

Figure 1: Structure of the ZIP file format

The common way of reading a ZIP file involves a series of steps. The first step is to search for the EOCDR by its 4-byte signature (`50 4b 05 06`), because the EOCDR is placed at the end of the file with a variable size. The EOCDR provides the position and size of the central directory and the number of CDHs. Then one can iterate through the CDHs and locate the corresponding LFHs at the positions specified in each CDH. The metadata for each file is then available and the data that follows the LFH can be decompressed.

ZIP files can be stored on a variety of storage medias, or transmitted in a data stream without being stored. Seeking the file and thus reading backward starting from the end of the file is not always feasible, so there is also an alternative "streaming" way of reading a ZIP file right from the beginning, using only the information in the LFHs to parse the ZIP file, either ignoring the central directory and the EOCDR or optionally checking their consistency with the LFHs later.

Since its first release in 1989, the ZIP file format has evolved a lot with many feature extensions. One of the most remarkable extensions is ZIP64 which utilize 64-bit integers to support file sizes larger than 4GiB. It consists of the ZIP64 extended information extra field for representing large file sizes, and the ZIP64 End Of Central Directory Record (ZIP64

EOCDR) with a ZIP64 End Of Central Directory Locator (ZIP64 EOCDL) to support a large central directory with more than 65535 entries. Besides the official extensions, there is also room for customized extensions through the extra fields in the CDH and LFH.

2.2 Inconsistent ZIP Parsing Behaviors

Although the ZIP file format was defined by PKWARE over thirty years ago, the specification document was casually written, leaving many important details unspecified and open to different interpretations. In 2015, ISO/IEC 21320-1 [4] was published to provide a formal standard for the ZIP file format. However, the standard only imposes several restrictions against the use of certain features, such as limiting the choices of compression methods, restricting the charset of filenames, and prohibiting the use of encryption, digital signature, patched data, and multi-volume spanning, without providing clarification for the previously unspecified details.

The ZIP file format is widely used for various purposes, and has been implemented in numerous applications and software libraries across a wide range of programming languages, each may have its own interpretation of the ZIP specification. Not only that the official specification is not clear enough, but the huge number of independent implementations also renders it impractical to name a de facto standard for ZIP. When it is hard to argue which one of two conflicting implementations is the correct one, an alternative solution is to reject inputs that lead to inconsistent outputs. However, programmers tend to follow the Postel's Law, "be conservative in what you do, be liberal in what you accept from others" [30], so parsers usually try their best to resolve malformed files instead of reporting errors. Consequently, it is common to see behavior discrepancies between ZIP parsers.

The first documented instance of ZIP parsing ambiguity vulnerability in the CVE list is CVE-2003-1154, where the attacker was able to bypass virus detection via malformed ZIP email attachments [3]. Over the past twenty years, prior work mainly focused on discovering new instances of virus detection bypassing. This kind of vulnerability occurs when the antivirus engine and the unarchive application used by the victim user parse the ZIP file differently. The contents of the ZIP file may appear innocent when parsed by the antivirus engine but malicious when extracted by the unarchive application. A few vulnerabilities involve other scenarios, such as APK signature verification [20] and Firefox add-on review [16].

3 Overview

3.1 Threat Model

In this paper, we consider an attacker who can craft a malicious ZIP file and deliver it to the victim. The attacker is not restricted to conventional archiving tools. They can arbitrarily manipulate the file at the byte level, even in ways that violate the ZIP specification. The malicious ZIP file will then be processed by different parsers with varying interpretations of the file contents, and these divergences can be exploited by the attacker to bypass security measures. We identified various real-world exploitation scenarios, including secure email gateway bypass, signature forgery, and content spoofing, which will be detailed in Section 6.

3.2 Motivating Example

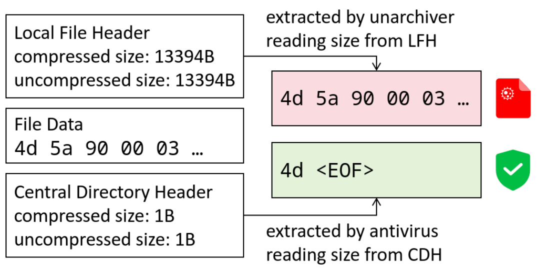

Figure 2: A motivating example to bypass secure email gateway via a ambigious ZIP file

In Fig. 2 we present a real-world example of bypassing secure email gateway using a malformed ZIP file. The ZIP file contains a malware payload as a file stored in the archive. The attacker manipulates the file size fields in the CDH corresponding to the malware, where the size of the file entry is set to one byte, so that some antivirus scanners that obtain the file size from the CDH would fail to identify the malware, including the ones used by mail.ru and inbox.lv. However, the size fields in the LFH are left untouched. Many ZIP unarchivers, including popular ones like WinRAR, 7-Zip, and Info-ZIP, use the file size fields in the LFH and thus are able to extract the malware. The victim user may trust the antivirus scanning report and execute the extracted malware, and then the victim's system will be infected by the malware.

This attack only manipulates the fields in the ZIP file and is independent of the specific content of the malware payload. No obfuscation, encryption, or transformation of the malware is required. So this attack is broadly applicable to any malicious file payload.

3.3 Research Questions

This paper answers the following research questions:

RQ1: How do we systematically identify inconsistencies between ZIP parsers? Prior work [16, 20, 23] has identified individual cases of ZIP parsing ambiguities. However, they rely on manual and ad hoc approaches for discovery. In this paper, we designed a mutation-based differential fuzzing tool ZIPDIFF that analyzes the discrepancies between the outputs of multiple ZIP parsers to identify parsing inconsistencies and to guide the fuzzing process. We combined grammar-aware mutations and byte-level mutations to generate the test samples. The ZIP file format has a complex structure and the field values often depend on each other. We carefully designed the mutation strategies to satisfy these requirements so that valid ZIP files are generated with a high probability. According to the test results, we classified the parsing ambiguities into three categories consisting of 14 distinct types along with root-cause analysis.

RQ2: What is the current prevalence of inconsistencies among real-world ZIP parsers? The ZIP file is not only one of the most popular archive formats, but also serves as the container for many other file formats. It has a long history and is now used virtually everywhere. As a result, there are many ZIP applications and libraries across different programming languages. In our study, we collected 50 different applications and libraries across 19 programming languages to discover ZIP parsing ambiguities comprehensively and measure the prevalence of inconsistencies among real-world ZIP parsers. The evaluation results show that almost all pairs of parsers are vulnerable to some parsing ambiguities.

RQ3: How can these inconsistencies be exploited in real-world products? We examined five scenarios where a ZIP file is processed by multiple entities, revealing how parsing inconsistencies between them can be exploited. In each scenario, we identified vulnerabilities in widely used real-world products and responsibly disclosed these issues to the vendors. Additionally, we proposed seven strategies to mitigate these vulnerabilities from various perspectives.

4 Design and Implementation

4.1 Workflow

To efficiently find discrepancies between ZIP parsers, we designed and implemented ZIPDIFF, our mutation-based black-box differential fuzzer for ZIP files. ZIPDIFF randomly generates and mutates ZIP files, and then feeds them to multiple ZIP parsers. The parsing outputs are used as criteria for selecting seeds for further mutations, feedback for the UCB-based mutation selection, and the indication of inconsistent parsing behaviors. The overall workflow is illustrated in Fig. 3.

4.2 Sample Generator & Mutator

To begin the fuzzing process, ZIPDIFF first generates some ZIP file samples as the initial corpus. The sample generator randomly selects the filenames and contents stored in the ZIP archive and ZIP parameters such as compression methods and ZIP feature set to build well-formed ZIP files. In fuzzing iterations, ZIPDIFF randomly mutates the test samples. There are two types of mutation strategies:

ZIP-Level Mutations. Some test samples are stored as structured ZIP files instead of raw bytes. The mutator is then able

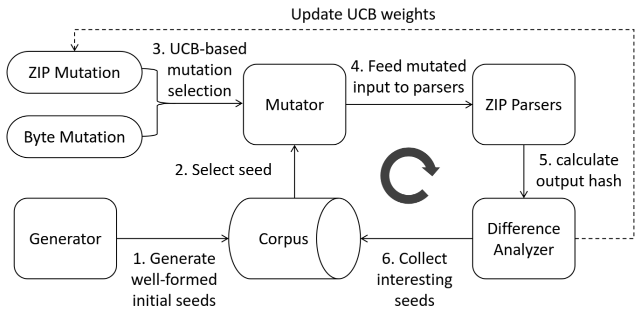

Figure 3: Workflow of ZIPDIFF

to locate and randomly mutate individual fields or a combination of multiple fields. We designed 46 mutation strategies of this type, covering every field of a legitimate ZIP file. These mutations are specially designed according to the semantic characteristics of different fields. For example, the compression method is either selected within a set of valid options with a high probability or randomly generated with a low probability. Some mutation strategies repair the inter-field dependencies such as the sizes and offsets so that more parsers are able to parse the mutated test samples successfully.

Byte-Level Mutations. To cover more edge cases, ZIPDIFF also employs byte-level mutations, including byte insertion, deletion, modification, duplication, splicing, and bit flipping. After byte-level mutations, the test sample will be stored as raw bytes without ZIP structures and ZIP-level mutations cannot be applied afterwards. Note that some ZIP-level mutations act like byte-level mutations but target individual ZIP fields, such as flipping the flag bits in LFH and CDH.

The mutator may apply multiple mutations in a single iteration to trigger ambiguities with a combination of mutations. Otherwise, the first mutation may be discarded if it does not trigger ambiguities on its own, as mutated samples are added back to the corpus only when they are found to be interesting.

4.3 Difference Analyzer

ZIPDIFF instructs each ZIP parser to read the sample ZIP file and extract it onto the file system. Each parser either successfully produces an output directory or reports an error. The difference analyzer computes a hash value of each successful output directory for subsequent comparisons, as illustrated in Algorithm 1 in Appendix A.

In preliminary experiments, we found that different parsers often process invalid characters in filenames inconsistently. However, inconsistencies in filenames are usually exploited only for specific file paths, such as `word/document.xml` for DOCX files. These specific paths usually do not contain invalid characters, so the inconsistency is only exploitable if the filenames are valid and controlled by the attacker. To filter out the non-exploitable filename inconsistencies, all filenames containing special or invalid characters are considered equal in the hashing process. Empty directories are also ignored in hash computation because they are not exploitable.

After obtaining these hash values, the analyzer compares each pair of parsers to see which pairs are inconsistent. Two parsers are considered inconsistent on a test sample if they both successfully extracted the sample file but the hash values differ. We do not regard a successful parser and a failing parser as inconsistent, because in practice inconsistencies are exploitable only when both parsers succeed in parsing the ZIP file. ZIPDIFF classifies the test samples into *interesting* samples and boring samples. A sample is *interesting* if it is not *covered* by any other seed in the corpus. We say that sample A is covered by sample B if A does not introduce any additional inconsistent parser pair or successful parser compared to B. The interesting sample detection algorithm is presented in Algorithm 2.

4.4 Mutation Strategy Selector

We implemented diverse mutation strategies targeting different fields in the ZIP file format. Because certain field mutations have a higher likelihood of triggering parsing ambiguities, the fuzzer should prioritize these high-impact strategies to enhance efficiency. However, accurately determining which strategies are most effective requires sufficient sampling. Each mutation strategy needs to be tested multiple times to reliably assess its probability of causing parsing ambiguities.

In summary, we are now facing the classical problem of maintaining balance between exploration and exploitation. To solve this problem, we treat the mutation strategy selection process as a multi-armed bandit problem and utilizes the Upper Confidence Bounds (UCB) formula [7] to choose the mutation strategies during the fuzzing process. The weight of each mutation strategy is defined as follows:

$$w_i := \frac{R_i}{N_i} + \sqrt{\frac{2\ln\sum_{j=1}^{K} N_i}{N_i}} \qquad (1)$$

where N_i is the number of times the mutation is used, R_i is the reward of the mutation gained from interesting samples, and K is the total number of mutation strategies. When a sample is mutated by multiple strategies in a single iteration, the usage count and reward are shared among them evenly.

In contrast to the original UCB algorithm, ZIPDIFF uses softmax (with a temperature parameter β) instead of argmax to select from the available options randomly and avoid using the same mutation strategy repeatedly:

$$\text{choice} := \underset{i\in[1,K]}{\text{weighted rand}} \left(\frac{e^{\beta w_i}}{\sum_{j=1}^{K} e^{\beta w_j}} \right) \qquad (2)$$

As opposed to the standard multi-armed bandit problem, the outcome of each mutation strategy changes over time, as the corpus evolves during the fuzzing process. To appreciate this property, ZIPDIFF decays the recorded usage count and reward over time by a rate of α in each iteration, so that recent evaluation results are weighted heavier than the old ones. The overall fuzzing process is illustrated in Algorithm 3.

5 Evaluation and Findings

5.1 Experiment Setup

To evaluate ZIPDIFF and measure the prevalence of ZIP parsing inconsistencies, we collected 50 ZIP parsers across 19 programming languages, including 4 applications and 46 programming libraries, as listed in Appendix B.

The applications are the most popular ones used by many end users, namely Info-ZIP, WinRAR, 7-Zip and its fork p7zip. For libraries, we use the official utility or example programs if available. When a library only provide low-level APIs like reading the file entries, we prefer wrapper libraries with high-level operations such as extracting the whole archive. For example, we use the `miniunzip` program provided by minizip itself, and we use the ZeroTurnaround ZIP library as a wrapper to test the ZIP classes provided by the Java standard library. When a library provides options to perform integrity checks for ZIP files, we enable these options.

Some libraries provide multiple sets of API to read ZIP files, usually a standard parsing method and a streaming one, as introduced in Section 2.1. In this case, we regard each set of API as a separate parser. We usually choose the latest stable version of each parser at the time of evaluation, with a few exceptions. For example, we include both yauzl v2 and yauzl v3, because a wrapper library `extract-zip` based on yauzl v2 contributes more than ten million weekly downloads, 80% of the yauzl weekly download count.

We check the dependency graphs and search for ZIP format parsing logic in the code of every parser to ensure that the parsers are substantially different and do not depend on each other, except in the aforementioned situations where a single library provide different APIs, when we include multiple versions of the same library, or when we evaluate a library through its wrapper library.

As we include a variety of parsers across many programming languages, to ensure maintainable code and reproducible evaluation, we configure each parser as a Docker image and run them in isolated environments. In particular, we use wine to run WinRAR in Docker on Linux to unify the evaluation process. As Docker invocation is expensive, we run test cases in batches, dispatching test cases in each batch inside the Docker containers. All parsers run concurrently, while some slow parsers also process multiple test cases in parallel to prevent becoming the bottleneck and ensure maximal utilization of available computing resources.

We conducted our experiments on a Linux server equipped with a 2GHz 112-core CPU and 944GB RAM.

5.2 Findings

Based on the fuzzing results, we analyze the sample files in the corpus and classify the identified ZIP parsing ambiguities as 14 distinct types and group them into three major categories: redundant metadata, file path processing, and ZIP structure positioning, as detailed in the following subsections.

In addtion to mutation-based fuzzing, we also constructed sample files for each variant of the ambiguities and tested the parsers against these samples. We summarize the number of inconsistency types between each parser pair in Appendix C. 1221 out of a total of 1225 pairs of parsers are affected by at least one type of ambiguity, demonstrating the prevalence of inconsistencies across ZIP parsers.

5.2.1 Redundant Metadata

Metadata for each file in a ZIP archive is typically stored in two locations, one in the local file header, and one in the central directory header. This design facilitates error recovery and allows streaming data processing, but it also leads to ambiguities when the metadata in different locations disagree. Some metadata are stored in extra locations besides LFH and CDH, such as the extra fields and the data descriptors, enlarging the room for ambiguities. In addition to identical metadata being stored in multiple locations, ambiguities can also occur when different metadata fields are capable of deriving the same piece of information.

Compression Method Confusion (A1). If the compression method specified in the CDH and the one specified in LFH are different, the parser may implicitly choose one from them. Although it is difficult or impractical to construct valid compressed data that can be successfully decompressed by multiple algorithms, ZIP allows storing a file without compression, using the "stored" compression method. An attacker can construct a ZIP file where the data is compressed but the compression method in either CDH or LFH is modified to "stored". In this way, parsers that select the correct compression method are able to extract the meaningful content, while the others that select the "stored" compression method will directly use the compressed data as output.

File Size Confusion (A2). Two types of file sizes are stored in ZIP files: the compressed size and the uncompressed size. The uncompressed size is redundant because it can be derived from the compressed data and the compression algorithm, but the parser may use it to truncate or pad the decompressed data. When the compression method is "stored", these two sizes should be identical, so the parser can choose either of them. Besides CDH and LFH, both compressed and uncompressed sizes can also be found in data descriptors and ZIP64 extended information extra fields if the respective features are enabled. Moreover, a single CDH or LFH may contain multiple ZIP64 extra fields. The size fields in the headers are set to `0xFFFFFFFF` when using ZIP64 and are set to zero when using data descriptors, which might be mistakenly used as the actual size by a parser not supporting these features. In summary, there are virtually an unlimited number of sources of file size information for the parser to choose from.

The integrity of files stored in a ZIP archive are protected by the CRC32 checksum. This integrity check is not enforced by many parsers, allowing attackers to manipulate the file data without worrying about the checksum. However, even if the integrity check is enforced, as CRC32 is not a cryptographic hash function, it is easy to pad a byte sequence with extra bytes while maintaining the CRC32 checksum unchanged [40], making the *File Size Confusion* exploitation more powerful.

Filename Confusion (A3). In addition to CDH and LFH, the filenames can also be stored in the Info-ZIP Unicode path extra field (hereafter abbreviated as UP). UP was designed to store a Unicode file path as an alternative to the ASCII file path. However, it does not enforce the presence of Unicode characters, and can be used to introduce a new source for the ASCII filename. UP can override the original filename field, but only a subset of parsers support this feature.

UP has three subfields, *version*, *name CRC32*, and *Unicode name*. The *version* field is reserved for incompatible changes in the future. The *name CRC32* field is the CRC32 checksum of the original filename, used to verify that the Unicode name is updated correspondingly when the original filename is changed. When there are multiple UPs in the extra fields, the parser has to select one of them. The selection strategy of the UP among multiple extra fields is usually implicitly implemented rather than intentionally designed. It can be quite complex, as there are many factors that can influence the selection process. We identified 6 edge cases: 1) The parser may select the first or the last UP in the extra fields. The extra field could be in the CDH or the LFH; 2) A UP with a *version* field not equal to 1 may be discarded. Some parsers discard only versions greater than 1, while others also discard version 0; 3) When the original filename does not match the *name CRC32*, the UP might be discarded; 4) In the CRC32 check, some parsers use the filename field in the CDH/LFH as the "original filename", while some others use the *Unicode name* from the previous UP; 5) When an invalid UP is discarded, the parser may either continue processing the remaining extra fields or stop. If it stops, the filename may be set to the last valid *Unicode name* or the original filename from the CDH/LFH; 6) There is a recently introduced language encoding flag in the CDH and LFH. This flag allows setting Unicode filenames directly in the original filename fields, deprecating the use of UP. When the flag is set, some parsers stop processing UP, while others still recognize UP.

Fake Directory (A4). Files stored in a ZIP archive can be either a regular file or a directory. There are two information sources to determine whether a file is a directory. The first source is whether the file path ends with a slash. Most parsers agree that a file path ending with a forward slash is a directory, except a few that do not think any file with a non-zero size is a directory. However, file paths that end with backslashes are not universally treated as directories. The second source is the *external file attributes* field in the CDH. This field is host-system dependent. There are different flag values representing a directory on MS-DOS and Unix. Parsers may rely on either or both of the flags on different systems. The *version made by* field indicates the host system. Some parsers determine the interpretation of *external file attributes* based on the host system. Some parsers only recognize the Unix directory flag when the host system is Unix, but the Go package `archive/zip` also recognizes the flag when the host system is OS X.

Fake Encryption (A5). When two parsers disagree about whether a file in a ZIP archive is encrypted, and the file data is actually unencrypted, only the parser that identifies it as unencrypted will successfully extract the file contents. This disagreement can arise from LFH-CDH confusion or from differences in parser support for file encryption. Furthermore, since ZIP archives typically encrypt either all files or none of them, some parsers give up processing the entire archive upon encountering an encrypted file. Consequently, if only the first file in an archive is encrypted, certain parsers will fail to extract any remaining files.

5.2.2 File Path Processing

A ZIP archive stores not only the contents of the files, but also the paths to these files. Discrepancies related to file paths usually enable attackers to effectively switch which file is at a specific path, thereby manipulate the contents of file formats that use ZIP as a contanier.

Duplicate Files (B1). When two or more files in a ZIP archive share the same path, and the parser is asked to retrieve the file at this path, it has to either intentionally or implicitly make a choice, where different parsers have divergent policies. Duplicate files also serve as a basis for other parsing ambiguities in this category. For example, suppose that two parsers A and B both select the last one among duplicate files, then the *Duplicate Files* ambiguity cannot be exploited on its own. However, with the help of some other parsing ambiguities, it would be possible to make parser A treat two files as both having the same path x, while parser B thinks the first file has path x but the second file has path y. When asked to retrieve the file with path x, parser A will see two files with path x and choose the second one, but parser B will choose the first file as it is the only file with the given path.

Invalid Characters (B2). Some characters are considered invalid in file paths, such as ASCII control characters, invalid Unicode, and "`"*:<>?|`" on Windows. Parsers may remove invalid characters or replace them with placeholders like "_" or "?". The valid character set and the processing mechanisms are different for each parser. Some parsers choose the text encoding based on the host system indicated by the *version made by* field. When encountering the null character, the file path string could be terminated in the middle.

Path Canonicalization (B3). There are multiple string representations of a single file path. For example, we can insert redundant slashes "`//`", replace forward slashes with backslashes, or use single dot "`.`" and double dots "`..`" to represent the current directory and the parent directory. The attacker can construct a ZIP archive (an ODT document) containing two files with paths `content.xml` and `./content.xml`. When retrieving the file at a certain path, different parsers may canonicalize the file paths inconsistently, so that some parsers think two files share the same path, while the other parsers think they are of different paths.

Case Sensitivity (B4). Most parsers compare file paths case-sensitively, but some parsers, especially those running on Windows, handle file paths in a case-insensitive manner. As a result, files with paths differing only in letter casing may be regarded as duplicates.

5.2.3 ZIP Structure Positioning

Before processing the file metadata and file paths, the first step in parsing a ZIP file is to determine the positions of the ZIP structures, i.e. to parse the EOCDR and the central directory. If parsers read the headers and file data from different positions, the parsing result may be completely different.

Streaming Parsing (C1). The standard mode for reading ZIP files is to use the information in the EOCDR to locate the central directory, and then locate individual LFHs through CDHs. Apart from that, LFHs can also be read sequentially from front to back in streaming mode, which introduces potential ambiguities. We describe several construction methods below. Although they utilize the same ambiguity, the diverse construction techniques make ambiguity detection difficult. If detection is not comprehensive, it may be bypassed.

1) *No corresponding CDH for LFH.* Parsers in standard mode usually only process LFHs referenced by CDHs, while parsers in streaming mode process all LFHs encountered during reading. If a file in the archive only has an LFH but no corresponding CDH, it is likely that only streaming parsing can read this file.

2) *Truncating the LFH stream.* In streaming mode, the parser reads consecutive LFHs until the end, where the end marker may be any non-LFH data, CDH, or EOCDR. However, in standard mode, LFHs can be placed anywhere, including after CDHs or even after the EOCDR. To avoid being detected as an anomaly, either the LFH can be put inside the comment field of a CDH or EOCDR, or the CDH or EOCDR that terminates the LFH stream is not the real one used in the standard mode.

3) *LFH desynchronization.* In streaming mode, an LFH must follow the previous entry's file data, while standard mode allows arbitrary LFH positioning. Thus, LFH positions can differ: a standard mode LFH might be placed where streaming mode expects file data. This may cause streaming parsers to misinterpret the LFH as part of the file data,

thus missing the LFH. This can be implemented by either including only these asynchronous LFHs (creating "holes" of unused bytes in standard mode) or both streaming and asynchronous LFHs (causing entry overlap in standard mode). Detection requires checking for such holes and overlaps.

4) *Data descriptor position.* ZIP's data descriptor feature, designed for streaming creation of ZIP files, places file sizes and checksum after the file data. In streaming parsing, without file size information in the LFH, the data descriptor's position (i.e., the end of file data) is uncertain. Parsers usually search for its signature to determine its position and may verify that its information matches the actual file data. A crafted file structure can make streaming parsers end file data at the data descriptor signature, while in standard mode, the signature is part of the file data. This can desynchronize LFH processing between the two modes. With a careful design, file sizes and other information can appear self-consistent in both modes, without causing holes or overlaps. To prevent this, streaming parsers should report an error when a data descriptor is used and the file size is unknown. Notably, compression methods like Deflate inherently record the file size, so the position is known even when using a data descriptor.

EOCDR Selection (C2). The EOCDR is placed at the end of the ZIP file with a variable size to store the ZIP file comment. It is possible to create a ZIP file containing multiple EOCDR signatures, all of which can be regarded as valid EOCDRs, since EOCDR signatures can be treated as a part of the comment field in an EOCDR. Most parsers scan the ZIP file backward from the end and choose the first encountered EOCDR signature. As a consistency check, some parsers verify that the comment length field either matches or is not bigger than the actual length of the comment. When the consistency check fails, the parser may either report an error or skip the EOCDR with an incorrect comment length and continue searching for the next one. In the latter case, the parser may choose a different EOCDR from other parsers that do not perform this consistency check.

The EOCDR selection policy used in libzip is unique and more complex. For an EOCDR, libzip first checks if there are inconsistencies such as incorrect comment length or mismatched field values in a pair of CDH and LFH. When no inconsistency is found, it calculates a "consistency" score based on the bounding byte range reached in the file, which is effective in detecting the outermost one among nested ZIP archives. Finally, the EOCDR with the highest "consistency" score is selected. With this policy, an attacker can create some inconsistency in the ZIP structure corresponding to the last EOCDR to trick libzip into using another EOCDR.

CDH Count Confusion (C3). The EOCDR provides information to parse the central directory, including the number of entries in the central directory. In particular, it provides two different forms of CDH count, one total CDH count, and one CDH count in the current disk, as ZIP files can be split into multiple disks. It also provides the size and the position of the central directory. All of these four fields can be used to determine the CDH count and may conflict with each other. We identfied 3 edge cases: 1) The total CDH count and current disk CDH count should match but may conflict when there is only a single disk; 2) The parser might read all CDHs inside the central directory, regardless of the values of the CDH count fields; 3) The parser may respect the central directory size field, or use all bytes until the EOCDR as the central directory.

In addition, as the CDH count fields are 16-bit, they are insufficient to represent large central directories with more than 65535 entries. As a workaround, some ZIP implementations store the actual CDH count modulo 65536 in EOCDR. Consequently, it is possible that two parsers both respect the same CDH count field, but one of them thinks there is only a single CDH, while the other thinks there are 65537 CDHs.

CD & LFH Offset Confusion (C4). The central directory is usually located by the offset field in the EOCDR. But some parsers assume that the EOCDR immediately follows the central directory with no gap between them, and thus use the central directory size to determine its position.

When the offset and the size fields in the EOCDR mismatch, it does not necessarily indicate that there is a gap between the central directory and the EOCDR. It is also possible that the ZIP file is padded with extra bytes at the beginning without adjusting the offset fields. For instance, this happens in self-extracting archives, where executable code is prepended to a ZIP archive to extract itself. In order to support this use case, some parsers not only assume that the EOCDR immediately follows the central directory, but the offset fields in the CDHs that point to the LFHs are also adjusted correspondingly. For example, if the offset field in EOCDR has value x, but the size field indicates that the actual offset to the central directory is $x+\delta$, then for a CDH with LFH offset value y, the parser will use $y+\delta$ as the actual offset to locate the LFH.

ZIP64 EOCD Processing (C5). The ZIP64 extension was developed to support ZIP files larger than 4GB and with more than 65535 files. It mainly consists of the ZIP64 extended information extra field, the ZIP64 end of central directory record (ZIP64 EOCDR), and the ZIP64 end of central directory locator (ZIP64 EOCDL). The ZIP64 EOCDL has a fixed size and contains the offset to the ZIP64 EOCDR whose size is variable. This design enables determined processing of the ZIP64 EOCDR without the need to search for its signature, but it also introduces more discrepancies. We identified the following ambiguities in ZIP64 EOCD processing: 1) The ZIP64 EOCDL can be located by a fixed offset from the regular EOCDR, or by searching for its signature; 2) The ZIP64 EOCDR can be located based on the ZIP64 EOCDL, or by searching for its signature; 3) It is unclear whether to use the ZIP64 EOCDR or the regular EOCDR. A parser may use the ZIP64 one whenever it is present, or only use the ZIP64 one when the fields in the regular one are set to `0xFFFFFFFF`; 4) If only a part of the fields in the regular EOCDR are set to `0xFFFFFFFF`, it is unclear whether to read all fields from the ZIP64 EOCDR, or to partly read from the regular one and mix fields from both ZIP64 and the regular EOCDR; 5) ZIP64 EOCDR provides an extra chance to suffer from the same issues as the regular EOCDR, including *CDH Count Confusion (C3)* and *CD & LFH Offset Confusion (C4)*.

5.3 Ablation Study

To evaluate our design decisions, we conducted an ablation study by comparing the performance of the following setups:

- **Full Setup:** As described in Section 4.
- **Argmax-Based UCB:** Use the original argmax-based UCB algorithm instead of our version that uses softmax for more randomness in mutation (Section 4.4).
- **Byte Mutation Only:** Use only the general byte-level mutations but not the ZIP-level mutations tailored for individual fields in the ZIP file format (Section 4.2).

We ran five 24-hour fuzzing sessions for each setup and plot the results in Fig. 4. The median numbers of discovered inconsistent parser pairs are 1197, 1183, and 1055 for the three setups, where the full setup outperforms the others. The softmax-based UCB algorithm allows exploring different mutation strategies in the same batch. The ZIP-level mutations can generate more valid samples and trigger more inconsistencies by focusing on mutating individual fields.

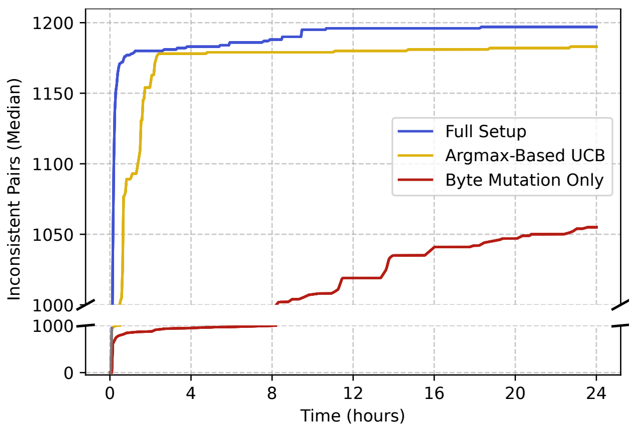

Figure 4: Median number of inconsistent parser pairs over time for each experiment setup.

6 Real-world Exploitations

In this subsection, we present five real-world scenarios where inconsistencies between ZIP parsers are weaponized. These scenarios target inconsistencies between different pairs of parsers in various applications, demonstrating the broad impact of ZIP parsing ambiguities.

Table 1: **Antivirus bypass results for email products.** The *Filename Confusion (A3)* type and the *File Path Processing (B)* category involve the filenames rather than the contents, so they have impacts on container formats but not antivirus bypass and are therefore excluded here.

●Vulnerable ○Not Vulnerable

Product	(A) Redundant Metadata				(C) ZIP Structure Positioning				
	1	2	4	5	1	2	3	4	5
Coremail	●	●	●	●	●	●	○	●	●
Gmail	○	○	○	○	●	●	○	○	●
iCloud	○	●	●	●	●	●	●	●	●
inbox.lv	●	●	○	●	○	●	●	●	●
mail.com	●	●	○	●	●	●	●	○	●
mail.ru	●	●	●	●	○	●	●	●	●
Naver	○	○	●	●	●	●	●	○	○
Outlook	●	●	○	○	●	●	○	●	●
Proton	●	●	○	●	●	●	●	●	●
Zoho	○	○	○	○	○	●	○	○	●

6.1 Secure Email Gateway Bypass

This scenario targets parsing inconsistencies between antivirus scanners and ZIP unarchivers. An attacker can craft a malicious ZIP file containing malware that can be extracted by a ZIP unarchiver but cannot be detected by the antivirus scanner. For example, the antivirus scanner might read truncated malware, treat compressed malware as uncompressed, regard the malware as a directory instead of a regular file, or assume that the malware is encrypted and give up processing.

For host-based antivirus software, this kind of bypass usually has limited impact since the malware is likely to be detected during on-access scanning when the user extracts the ZIP archive. However, it is more critical in remote environments such as secure email gateways, where host-based defense might be absent in the end user's system. For instance, Gmail will scan attachments in the emails and display a note "Scanned by Gmail" besides the attachments. When a user receives an email with an attachment passing the secure email gateway, they may trust its content based on the scanning result and open it through a ZIP unarchiver that is able to extract the malware, and then the system will be infected.

We tested various ZIP parsing ambiguity constructions on popular email products that provide virus scanning services and allow us to register free testing accounts. We first send an email to our testing account with a safe ZIP attachment to verify that it can be delivered to the inbox and another email with a well-formed ZIP file containing malware to verify that it is rejected. Then we send emails with malformed ZIP files with malware for antivirus bypass testing. All tested email products are vulnerable to some construction methods, as listed in Table 1.

6.2 Office Document Content Spoofing

This scenario targets parsing inconsistencies between different office applications, including office suites like Microsoft Office and LibreOffice, as well as Web applications such as plagiarism checkers and AI assistant services (e.g. ChatGPT, Claude, DeepSeek, etc.) that process office documents.

An office document is a ZIP file containing some XML files. The document content is stored in the file with a certain file path, such as `word/document.xml` in DOCX files and `content.xml` in ODT files. An attacker can construct an office document that is parsed inconsistently by different applications, so that these applications will see different document content.

The security implications of content spoofing depend on the specific use cases. A representative case is plagiarism checker bypassing, where an unethical student can construct a document such that plagiarized content is displayed in the office suite but hidden from the plagiarism checker. The supervisor reads the document in the local office suite application and sends the document to a remote service to detect plagiarism. The supervisor will read the plagiarized content but may trust its originality based on the plagiarism checker report.

Case Study 1. The plagiarism checker provided by China National Knowledge Infrastructure locates the XML file case-insensitively and picks the last one among duplicate files. In contrast, Libreoffice ignores `WORD/DOCUMENT.XML` in uppercase, vulnerable to *Case Sensitivity (B4)*. WPS Office on Windows locates the file case-insensitively but selects the first one among duplicate files, vulnerable to *Duplicate Files (B1)*.

Case Study 2. A dishonest student can construct a document with the structure shown in Fig. 5. The two columns of the table represents two possible parsing results, where each row in both columns represents the same bytes in the ZIP file. It exploits *ZIP64 EOCD Processing (C5)*, where ZIP64 EOCDR and EOCDL are present in the ZIP file, but the fields in the regular EOCDR are not set to `0xffffffff` as required in the specification. In this case, Microsoft Office, LibreOffice, and WPS Office will ignore the ZIP64 EOCDR and use the central directory corresponding to the regular EOCDR, thus reading the `word/document.xml` with plagiarism, as the right column in Fig. 5, where the extra `word/document.xml` and ZIP64 EOCD are treated as a CDH comment field by setting the comment length field in the last CDH. However, the PapersOwl plagiarism checker will recognize the ZIP64 EOCDR and read the `word/document.xml` without plagiarism, as the left column in Fig. 5, so it will report no plagiarism in the document. In addition, the Grammarly plagiarism checker works in the *Streaming Parsing (C1)* mode. It also parses the document as the left column, but it reads the LFHs one by one instead of relying on the ZIP64 or regular EOCDR.

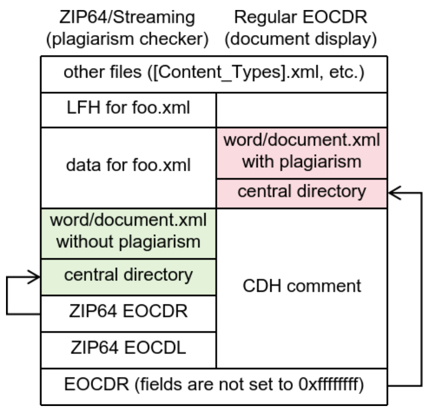
ZIP64/Streaming (plagiarism checker)
Regular EOCDR (document display)
other files ([Content_Types].xml, etc.)
LFH for foo.xml
data for foo.xml
word/document.xml with plagiarism
central directory
word/document.xml without plagiarism
central directory
CDH comment
ZIP64 EOCDR
ZIP64 EOCDL
EOCDR (fields are not set to 0xffffffff)

Figure 5: Plagiarism checker bypass example

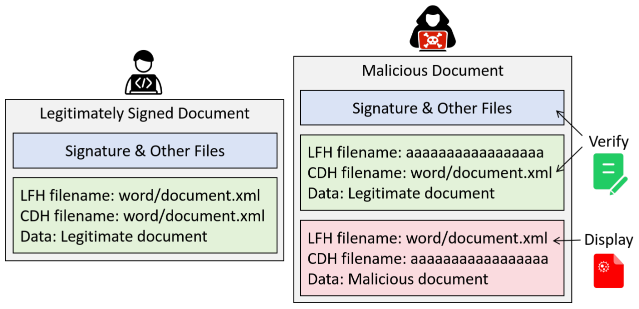
Legitimately Signed Document
Signature & Other Files
LFH filename: word/document.xml
CDH filename: word/document.xml
Data: Legitimate document
Malicious Document
Signature & Other Files
LFH filename: aaaaaaaaaaaaaaaaa
CDH filename: word/document.xml
Data: Legitimate document
LFH filename: word/document.xml
CDH filename: aaaaaaaaaaaaaaaaa
Data: Malicious document
Verify
Display

Figure 6: LibreOffice document signature forgery example

6.3 LibreOffice Document Signature Forgery

This scenario targets parsing inconsistency between the signature verifier and the document viewer of LibreOffice. Suppose that the attacker has obtained a legitimately signed document. With the legitimate signature and the corresponding document content, the attacker can construct a malicious document. Due to parsing inconsistency, the signature verifier sees the original legitimate content and reports that the signature is valid. However, the document viewer displays the manipulated content, leading to signature forgery.

Different components in a single application usually use the same parser and are thus not vulnerable to parsing discrepancies. In this specific case, LibreOffice indeed uses the same parser for signature verification and document display. However, the parser has a normal mode and a recovery mode, where the normal mode detects inconsistencies in the ZIP file but the recovery mode ignores errors and works in the streaming parsing mode. When the parser in the normal mode finds a document to be corrupted, it will use the recovery mode to display the document, but the signature validation still uses the normal mode and the valid status remains unchanged.

Case Study. Based on a signed document, the attacker modifies the original `word/document.xml` entry to change its filename field in the LFH and adds a new entry with filename field `word/document.xml` in LFH to exploit the *Filename Confusion (A3)* ambiguity, as illustrated in Fig. 6. The signature verifier will validate the signature against the file with `word/document.xml` as CDH filename, but the document displayer will show the file with `word/document.xml` as LFH filename after entering the recovery mode.

6.4 Spring Boot Nested JAR Signature Forgery

This scenario targets parsing inconsistency between two JAR parsers used by the `NestedJarFile` class in Spring Boot Loader. A custom ZIP parser `ZipContent` is implemented to read the content of a JAR file, which operates in the standard ZIP parsing mode. However, the `JarInputStream` class provided by Java is utilized to verify its signature, with the *Streaming Parsing (C1)* mode. Therefore, with a legitimately signed nested JAR file, the attacker can forge a new one with arbitrary contents and pass the signature verification.

6.5 VS Code Extension ID Impersonation

This scenario targets parsing inconsistency between the VS Code extension Marketplace server and the VS Code client. VS Code extension packages are ZIP files. The Marketplace ensures that authors can publish extensions only within their own namespaces. The publisher and extension ID are recorded in the `extension.vsixmanifest` file. The server and the client are vulnerable to *Filename Confusion (A3)*, allowing the attacker to construct an extension package such that the server and the client read different `extension.vsixmanifest` files and thus different publishers, and then the attacker will be able to circumvent the namespace isolation rule.

Case Study. Suppose that the attacker owns the namespace `attacker` and the target extension is `bob.foo`. The attacker can publish a malicious extension exploiting the Unicode path extra field such that the Marketplace server recognizes its ID as `attacker.bar` but the VS Code client reads its ID as `bob.foo`, as demonstrated in Fig. 7. Consequently, if a victim user installs the malicious extension, it will impersonate the target extension and replace the originally installed one.

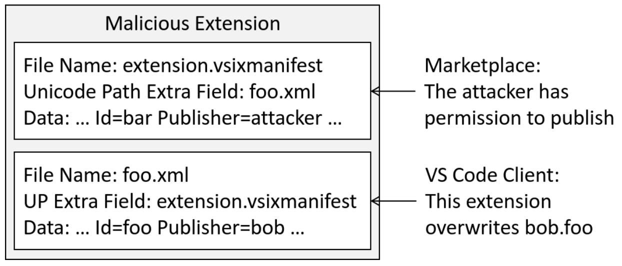
Malicious Extension
File Name: extension.vsixmanifest
Unicode Path Extra Field: foo.xml
Data: … Id=bar Publisher=attacker …
File Name: foo.xml
UP Extra Field: extension.vsixmanifest
Data: … Id=foo Publisher=bob …
Marketplace: The attacker has permission to publish
VS Code Client: This extension overwrites bob.foo

Figure 7: VS Code extension ID impersonation example

7 Discussion

7.1 Responsible Disclosure

We have reported our findings to the affected vendors and coordinated in addressing these issues. We received many acknowledgments and bounty awards, as summarized below.

7.1.1 Email Services

- **Gmail**: acknowledged our report, rated the vulnerability as medium severity with a bug bounty reward of $1337, and deployed defense against the reported issues.
- **Coremail**: acknowledged our report, rated the vulnerability as medium severity with a bug bounty reward of about $400, and arranged to fix the reported issues.
- **Zoho**: acknowledged our report, rated the vulnerability as medium severity with a bug bounty reward of $200 and deployed defense against the reported issues.
- **Outlook**: acknowledged our report and rated the vulnerability as low severity with an entry in the MSRC Online Services Acknowledgements. They also improved their antivirus scanning logic according to our report.
- **Proton Mail, Naver, mail.com, and mail.ru**: our reports were acknowledged but not eligible for bounty rewards.
- **iCloud**: said they were still investigating our report.
- **inbox.lv**: has not responded to our report yet.

7.1.2 Office Applications

- **LibreOffice**: promptly acknowledged our reports on both content spoofing and signature forgery. They carried out an in-depth conversation with us and inspired us to develop some novel bypass techniques against their proposed patches. They fixed these issues and assigned CVE-2024-7788 for the signature forgery vulnerability.
- **cnki.net**: acknowledged our report on plagiarism scanning bypass and planned to fix the issue.

7.1.3 ZIP Parsers

- **Go archive/zip**: acknowledged our report, assigned CVE-2024-24789, and changed their implementation to reject truncated EOCDR comments.
- **libzip**: acknowledged our report and changed their unique implementation of EOCDR selection to align with other parsers in the lax mode and report an error in the strict mode. They have also implemented stricter consistency checks according to our suggestions.

7.1.4 Others

- **Spring Boot**: acknowledged our report, assigned CVE-2024-38807, and fixed the vulnerability.
- **Open VSX**: acknowledged our report and implemented a check for malicious extension packages.

7.2 Mitigation

We propose seven mitigation strategies with different concerns and trade-offs. Developers can choose the most appropriate mitigation according to the specific situation.

Use the same parser. If all parsers used in a workflow are controlled by the same party, then the best solution is to use the same parser in all places. However, the parsers are usually controlled by multiple parties, where no single party can control the parsers used by others, so this simple solution has limited applications.

On-access scanning. Antivirus software typically uses on-access scanning to compensate for the shortcomings of directly scanning archive files. This is not only a solution for antivirus, but the same concept also applies to other scenarios, where a component can use the parsing result of another component instead of parsing the ZIP file again. It can be regarded as a special way to enforce using the same parser. For example, a plagiarism checker may work directly inside an office suite, using the parsing result from the office suite.

Normalize the ZIP file. To exploit ZIP parsing ambiguities, the attacker usually needs to carefully manipulate the fields of a ZIP file, which cannot be achieved by a regular ZIP archiver. Most ambiguities will disappear if the ZIP file is first extracted and then repacked. Therefore, if we care about only the contents but not the integrity of the whole ZIP file, we can normalize the ZIP file by extracting and repacking it before processing.

Identify ambiguous patterns in ZIP files. Malformed ZIP files can be identified by special patterns, such as unused bytes and conflicting field values in CDH and LFH. For instance, libzip provides a `CHECKCONS` flag with intensive consistency checks, and LibreOffice warns users of malformed document files. A service may reject or report all malformed ZIP files without affecting legitimate users, as malformed files are usually intentionally crafted. However, the identification of ambiguous patterns relies on existing knowledge, and previously unknown ambiguities are hard to detect. In addition, malformed ZIP files also have legitimate use cases, such as self-extracting files with prepended executable code and APK with signature data before the central directory.

Incorporate different parsing logics. Instead of relying on some pre-defined patterns, a service can identify ambiguous ZIP files by incorporating multiple parsers to see if they produce consistent outputs. This can potentially detect previously unknown ambiguities. However, a large number of parsers are needed to cover all ambiguities, and it may consume a large amount of system resources to extract an archive multiple times. An alternative approach is to combine multiple parsing logics or try all possible choices in a single parser. For instance, Gmail identifies all CDHs and the corresponding files, even if they are overlapped or outside of the central directory.

Fix unique parsing behaviors. Parsing ambiguities can also be mitigated by making parsers behave consistently. Since the ZIP specification is vague and lacks many important details, there is no de facto standard, and it is usually infeasible to determine which parsers are correct when the implementations are inconsistent. It is challenging to collaboratively fix inconsistencies without the presence of a standard, but we can address the outlier behaviors where a few parsers behave differently from the majority of other parsers.

Better file format design. If we were able to redesign the ZIP file format or to design a new archive format, we could learn from the history and design a better file format:

- Each part of a format must be unambiguously located. It is a bad idea to rely on fragile signature searching.
- Conflicting data resolution should be clearly defined, ideally by avoiding redundant data in the first place.
- Leave room for backward-compatible feature extensions. Make it clear whether an extension is enabled or not.
- Fields that are allowed to be silently ignored should not contain security-sensitive data. For example, the extra fields in ZIP should not contain filenames and sizes.

7.3 Limitation

We chose blackbox fuzzing so that we can support more parsers and programming languages to uncover more ambiguities. As a blackbox fuzzer, ZIPDIFF only receives the parsing outputs as feedback and does not utilize greybox feedback like code coverage. It might generate more sophisticated test samples with cross-language coverage-guided fuzzing.

ZIPDIFF instructs the parsers to extract ZIP archives onto the file system in order to unify the testing process and provide easier parser integration. However, the output on the file system might not match the internal parsing result exactly. For example, duplicate files may be overwritten either in the internal state or when writing to the file on the disk. It may obtain more accurate results if the internal states are recorded.

Although we only identified vulnerabilities in five scenarios, the ZIP file format is also used in other security-sensitive scenarios. Further research may extend our results to broader fields. For instance, the 3MF data format used in 3D printing is based on ZIP and vulnerable to UI spoofing attacks [33]. While we focus on plagiarism checkers for the office document content spoofing scenario, it is also interesting to see whether this can be used for indirect prompt injection attacks against large language models.

Besides ZIP, other archive formats like TAR also suffer from parsing ambiguities [19]. However, ZIP is structurally more complex, consisting of LFHs, CDHs, and EOCDR. In contrast, TAR uses a simpler linear header structure, supports fewer extensions, and separates archiving from compression. We focus on ZIP in this paper because its complexity makes it more prone to semantic gaps. ZIPDIFF could be extended to handle other archive formats by replacing the ZIP-level mutations with other format-specific mutation strategies. We leave this extension as future work.

8 Related Work

8.1 Semantic Gaps and Differential Testing

There is a rich literature on semantic gaps in various aspects of network security, including TLS [8,10,13,21,37,41], RPKI [24], QUIC [31], HTTP [11,18,35,44,50], CSP [46], URL [5, 32, 36, 39, 45, 47], HTML [22], JSON [25], and Email [12,43,49]. These works use various methods to analyze semantic gaps, ranging from manual, ad-hoc testing to blackbox and greybox differential fuzzing.

Petsios et al. developed a domain-independent differential testing framework called NEZHA [28] and demonstrated its effectiveness by uncovering parsing discrepancies in ELF, XZ, PDF, and TLS. Our fuzzer ZIPDIFF determines interesting seeds by two metrics *ok* and *incons*, as illustrated in Algorithm 2. The *ok* metric is essentially the *output* δ-*diversity* proposed by Petsios et al., if we only care about the success status but not the output contents. Since ZIP parsers produce complex file trees as outputs, it is impractical to use the entire output contents as the output diversity. To utilize the output contents in fuzzing guidance, ZIPDIFF uses pairwise equality as the *incons* metric.

Zheng et al. [50] incorporated the UCT-Rand algorithm to select grammar nodes in their generation-based fuzzer REQSMINER. In contrast, our mutation-based fuzzer ZIPDIFF utilizes UCB to guide mutation strategy selection. We use the softmax function as a balance between the original argmax-based UCB and UCT-Rand.

8.2 ZIP Parsing Ambiguities

As early as 2008, Alvarez and Zoller presented a talk [34, 51] on antivirus evasion based on parsing discrepancies of various archive formats, including ZIP, RAR, and CAB. They revisited these issues in 2020 [52] and found that many antivirus engines are still vulnerable. Vuksan et al. gave a similar talk in 2010 [42], revealing more bypass techniques. Coldwind also presented talks [14, 15] that summarized attack surfaces of the ZIP file format including parsing ambiguities.

Jana and Shmatikov [19] proposed two types of antivirus bypass attacks based on file parsing ambiguities: the *Chameleon* attack exploits conflicting file type detection and the *Werewolf* attack exploits inconsistent parsing logic of the same file type. Their work suggested that many file types are vulnerable to both attacks, including archive formats like ZIP and other formats like ELF.

Panakkal [26] summarized several vulnerabilities on malformed APK files, and constructed a ZIP file that can be recognized as multiple container formats by mixing files related to different formats.

Two more vulnerabilities are also caused by ZIP parsing ambiguities. One [16] exploits inconsistencies between parsers used in the Mozilla Firefox add-on review pipeline to submit malicious add-on that appears benign at review time. The other [23] exploits a custom signature mechanism utilizing the file comment field in the EOCDR to bypass signature verification in firmware updates.

In summary, there is a lack of systemantic research on ZIP parsing ambiguities, with most previous studies focusing on individual vulnerabilities. Our work provided the first systematic synthesis of existing knowledge, developed a differential fuzzer, uncovered additional ambiguity classes, extended construction techniques, delivered an up-to-date evaluation of 50 parsers across 19 programming languages, and identified vulnerabilities in novel scenarios.

To the best of our knowledge, among the 14 ambiguity types, ten of them are discovered or extended by us with novel variants and techniques to cause more inconsistencies and bypass some checks. Details are listed in Table 2.

Table 2: Novelty of the ZIP ambiguity types
○ New ambiguity discovered by us – Known ambiguity

Type	Novel	New type, variant, or technique
A1	–	-
A2	○	Multiple ZIP64 & Fixing CRC32
A3	○	Unicode path extra field techniques
A4	○	Filename ending with backslash & Host system support in file attributes
A5	○	CDH vs LFH inconsistency for the “encrypted” general purpose flag
B1	–	-
B2	○	Special characters besides null byte & Affected by the host system field
B3	○	The entire type is novel
B4	–	-
C1	○	Multiple construction techniques to bypass inconsistent ZIP file checking
C2	○	Coldwind [15] mistakenly stated that libzip selected the first EOCDR. We identified the real mechanism used by libzip and some other parsers.
C3	○	Total vs current disk CDH count
C4	–	-
C5	○	The entire type is novel

8.3 Other ZIP Attacks

Besides parsing ambiguities, the ZIP file format also suffers from other attacks, such as *ZIP bomb* and *ZIP Slip*.

The data amplification attack, known as *ZIP bomb*, exhausts remote servers' resources by highly-compressed ZIP files. Pellegrino and Balzarotti [27] investigated the use of data compression in network services and analyzed relevant pitfalls and vulnerabilities. Canet et al. [9] focused on decompression quines, archives that decompress to themselves, and their impact on antivirus engines. Fifield [17] constructed a ZIP bomb that reaches a compression ratio of over 28 million by overlapping entries in a ZIP file, without the need of nested archives or uncommon compression algorithms. Their work also highlighted the compatibility issues among ZIP parsers, although from a perspective of finding universally working ZIP bombs rather than exploiting these inconsistencies.

ZIP unarchivers are also vulnerable to path traversal attacks, where files with malicious paths (`../`) are extracted outside of the target directory. The earliest instances of such vulnerabilities on the CVE list are CVE-2001-1268 [1] and CVE-2001-1269 [2]. Years later, the Synk security team identified many ZIP applications were vulnerable and branded the vulnerability as *ZIP Slip* [38].

Mitigation of path traversal can lead to inconsistencies in *Path Canonicalization (B3)*. For example, some parsers remove `../` in the file paths, while others resolve them and check whether the final result is inside the target directory. A file path `foo/../bar` is transformed to `foo/bar` in the former, but `bar` in the latter.

9 Conclusion

This paper presented the ZIPDIFF differential fuzzing tool and evaluated it on 50 ZIP parsers across 19 languages. The result revealed that almost all pairs of ZIP parsers are inconsistent. We summarized ZIP parsing ambiguities as 14 distinct types in three categories with different root-causes. These ambiguities can be exploited in various real-world scenarios as ZIP is used in numerous applications. The vulnerabilities can be mitigated by incorporating suitable defense strategies.

By examining ZIP parsing ambiguities as a specific instance of the broader problem on semantic gaps, our work highlights the critical need for rigorously defined file formats and consistent parser implementations. We hope this study not only inspires the community to identify and reduce discrepancies between ZIP parsers and to address the relevant vulnerabilities, but also raises general awareness of the security implications of semantic gaps.

Acknowledgments

We sincerely thank all anonymous reviewers for their insightful and constructive feedback that greatly helped improve the paper and the artifacts. This work is supported by the National Natural Science Foundation of China (grant #62272265).

Ethics Considerations

We have responsibly disclosed the identified vulnerabilities to the affected vendors, as listed in Section 7.1.

In the email product tests, we sent the emails with malicious attachments from our own server to our own email product accounts. We did not send malicious emails to any other user or from these email products. We throttled the email-sending frequency to avoid excessive pressure on the products.

In the plagiarism checker tests, we uploaded the malicious testing documents to the plagiarism detection services using our own account when an account was required. We did not send these malicious documents to other people.

In the VS Code extension tests, we published the testing VS Code extension on the marketplace with an explicit warning in the extension description that it is used for testing purposes only. It also does not contain any destructive functions. Only a popup message is used to indicate that the modified version of the extension is running.

Open Science

We share the artifacts on Zenodo [48] and GitHub[2] for the following components: the ZIPDIFF differential fuzzer (Section 4) along with ablation study options (Section 5.3), the Docker images of the tested ZIP parsers (Table 3), construction of ambiguous ZIP file samples (Section 5.2), and utility scripts to reproduce the results.

References

[1] CVE-2001-1268, 2001. https://nvd.nist.gov/vuln/detail/CVE-2001-1268.

[2] CVE-2001-1269, 2001. https://nvd.nist.gov/vuln/detail/CVE-2001-1269.

[3] CVE-2003-1154, 2003. https://nvd.nist.gov/vuln/detail/CVE-2003-1154.

[4] ISO/IEC 21320-1:2015(E) Information Technology — Document Container File — Part 1: Core, 2015.

[5] Dashmeet Kaur Ajmani, Igibek Koishybayev, and Alexandros Kapravelos. yoU aRe a Liar://A Unified Framework for Cross-Testing URL Parsers. In *2022 IEEE Security and Privacy Workshops (SPW)*, pages 51–58, San Francisco, CA, USA, May 2022. IEEE. https://doi.org/10.1109/SPW54247.2022.9833883.

[2]https://github.com/ouuan/ZipDiff

[6] Abhishek Arya, Oliver Chang, Jonathan Metzman, Kostya Serebryany, and Dongge Liu. OSS-Fuzz. https://github.com/google/oss-fuzz.

[7] Peter Auer, Nicol O Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47:235–256, May 2002. https://doi.org/10.1023/A:1013689704352.

[8] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *2014 IEEE Symposium on Security and Privacy*, pages 114–129, San Jose, CA, May 2014. IEEE. https://doi.org/10.1109/SP.2014.15.

[9] Margaux Canet, Amrit Kumar, Cédric Lauradoux, Mary-Andréa Rakotomanga, and Reihaneh Safavi-Naini. Decompression Quines and Anti-Viruses. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 23–34, Scottsdale Arizona USA, March 2017. ACM. https://doi.org/10.1145/3029806.3029818.

[10] Chu Chen, Pinghong Ren, Zhenhua Duan, Cong Tian, Xu Lu, and Bin Yu. SBDT: Search-Based Differential Testing of Certificate Parsers in SSL/TLS Implementations. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 967–979, Seattle WA USA, July 2023. ACM. https://doi.org/10.1145/3597926.3598110.

[11] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1516–1527, Vienna Austria, October 2016. ACM. https://doi.org/10.1145/2976749.2978394.

[12] Jianjun Chen, Vern Paxson, and Jian Jiang. Composition Kills: A Case Study of Email Sender Authentication. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2183–2199. USENIX Association, August 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-jianjun.

[13] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 793–804, Bergamo Italy, August 2015. ACM. https://doi.org/10.1145/2786805.2786835.

[14] Gynvael Coldwind. Ten thousand traps: ZIP, RAR, etc., 2013. https://gynvael.coldwind.pl/?id=523.

[15] Gynvael Coldwind. Ten thousand security pitfalls: The ZIP file format, 2018. https://gynvael.coldwind.pl/?id=682.

[16] David Fifield. Ambiguous Zip Parsing Allows Hiding Add-on Files from Linter and Reviewers, April 2019. https://www.bamsoftware.com/sec/mozilla/#1534483.

[17] David Fifield. A better zip bomb. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association. https://www.usenix.org/conference/woot19/presentation/fifield.

[18] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1805–1820, Virtual Event Republic of Korea, November 2021. ACM. https://doi.org/10.1145/3460120.3485384.

[19] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *2012 IEEE Symposium on Security and Privacy*, pages 80–94, San Francisco, California, USA, 2012. IEEE. https://doi.org/10.1109/SP.2012.15.

[20] Jay Freeman. Exploit (& Fix) Android "Master Key". https://www.saurik.com/masterkey1.html.

[21] Dan Kaminsky, Meredith L Patterson, and Len Sassaman. Pki layer cake: New collision attacks against the global x. 509 infrastructure. In *International Conference on Financial Cryptography and Data Security*, pages 289–303, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14577-3_22.

[22] David Klein and Martin Johns. Parse Me, Baby, One More Time: Bypassing HTML Sanitizer via Parsing Differentials. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 203–221, San Francisco, CA, USA, May 2024. IEEE. https://doi.org/10.1109/SP54263.2024.00177.

[23] Daniel Komaromy and Lorant Szabo. UnZiploc: A bug hunter's journey from 0- click to platform compromise, 2022. https://labs.taszk.io/articles/post/unziploc/.

[24] Donika Mirdita, Haya Shulman, Niklas Vogel, and Michael Waidner. The CURE to Vulnerabilities in RPKI Validation. In *Proceedings 2024 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2024. Internet Society. https://doi.org/10.14722/ndss.2024.241093.

[25] Jonas Möller, Felix Weißberg, Lukas Pirch, Thorsten Eisenhofer, and Konrad Rieck. Cross-Language Differential Testing of JSON Parsers. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1117–1127, Singapore Singapore, July 2024. ACM. https://doi.org/10.1145/3634737.3657003.

[26] Gregory R Panakkal. Leaving our zip undone: How to abuse zip to deliver malware apps, 2014. https://www.virusbulletin.com/virusbulletin/2015/03/paper-leaving-our-zip-undone-how-abuse-zip-deliver-malware-apps.

[27] Giancarlo Pellegrino, Davide Balzarotti, Stefan Winter, and Neeraj Suri. In the compression Hornet's nest: A security study of data compression in network services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 801–816, Washington, D.C., August 2015. USENIX Association. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/pellegrino.

[28] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632, San Jose, CA, USA, May 2017. IEEE. https://doi.org/10.1109/SP.2017.27.

[29] PKWARE Inc. APPNOTE.TXT - .ZIP File Format Specification, 2022. https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT.

[30] Jon Postel. DoD standard Transmission Control Protocol. RFC 761, January 1980. https://www.rfc-editor.org/info/rfc761.

[31] Gaganjeet Singh Reen and Christian Rossow. DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies for QUIC. In *Annual Computer Security Applications Conference*, pages 332–344, Austin USA, December 2020. ACM. https://doi.org/10.1145/3427228.3427662.

[32] Joshua Reynolds, Adam Bates, and Michael Bailey. Equivocal URLs: Understanding the Fragmented Space of URL Parser Implementations. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, volume 13556, pages 166–185, Cham, 2022. Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-17143-7_9.

[33] Jost Rossel, Vladislav Mladenov, and Juraj Somorovsky. Security Analysis of the 3MF Data Format. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 179–194, Hong Kong China, October 2023. ACM. https://doi.org/10.1145/3607199.3607216.

[34] Sergio Alvarez and Thierry Zoller. The Death of AV Defense in Depth? - revisiting Anti-Virus Software, 2008.

[35] Kaiwen Shen, Jianyu Lu, Yaru Yang, Jianjun Chen, Mingming Zhang, Haixin Duan, Jia Zhang, and Xiaofeng Zheng. HDiff: A Semi-automatic Framework for Discovering Semantic Gap Attack in HTTP Implementations. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–13, Baltimore, MD, USA, June 2022. IEEE. https://doi.org/10.1109/DSN53405.2022.00014.

[36] Taiga Shirakura, Hirokazu Hasegawa, Yukiko Yamaguchi, and Hajime Shimada. Potential Security Risks of Internationalized Domain Name Processing for Hyperlink. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1092–1098, Madrid, Spain, July 2021. IEEE. https://doi.org/10.1109/COMPSAC51774.2021.00149.

[37] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 521–538, San Jose, CA, USA, May 2017. IEEE. https://doi.org/10.1109/SP.2017.46.

[38] Snyk. Zip Slip Vulnerability, 2018. https://security.snyk.io/research/zip-slip-vulnerability.

[39] Daniel Stenberg. My URL isn't your URL, May 2016. https://daniel.haxx.se/blog/2016/05/11/my-url-isnt-your-url/.

[40] Martin Stigge, Henryk Plötz, Wolf Müller, and Jens-Peter Redlich. Reversing CRC – Theory and Practice, 2006.

[41] Cong Tian, Chu Chen, Zhenhua Duan, and Liang Zhao. Differential Testing of Certificate Validation in SSL/TLS Implementations: An RFC-guided Approach. *ACM Transactions on Software Engineering and Methodology*, 28(4):1–37, October 2019. https://doi.org/10.1145/3355048.

[42] Mario Vuksan, Tomislav Pericin, and Brian Karney. Hiding in the Familiar: Steganography and Vulnerabilities in Popular Archives Formats, 2010.

[43] Chuhan Wang, Yasuhiro Kuranaga, Yihang Wang, Mingming Zhang, Linkai Zheng, Xiang Li, Jianjun Chen,

Haixin Duan, Yanzhong Lin, and Qingfeng Pan. BreakSPF: How Shared Infrastructures Magnify SPF Vulnerabilities Across the Internet. In *Proceedings 2024 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2024. Internet Society. https://doi.org/10.14722/ndss.2024.23113.

[44] Qi Wang, Jianjun Chen, Zheyu Jiang, Run Guo, Ximeng Liu, Chao Zhang, and Haixin Duan. Break the Wall from Bottom: Automated Discovery of Protocol-Level Evasion Vulnerabilities in Web Application Firewalls. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 185–202, San Francisco, CA, USA, May 2024. IEEE. https://doi.org/10.1109/SP54263.2024.00129.

[45] Xianbo Wang, Wing Cheong Lau, Ronghai Yang, and Shangcheng Shi. Make Redirection Evil Again: URL Parser Issues in OAuth, 2019.

[46] Seongil Wi, Trung Tin Nguyen, Jihwan Kim, Ben Stock, and Sooel Son. DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing. In *Proceedings 2023 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2023. Internet Society. https://doi.org/10.14722/ndss.2023.24200.

[47] Qilang Yang, Dimitrios Damopoulos, and Georgios Portokalidis. WYSISNWIV: What You Scan Is Not What I Visit. In Herbert Bos, Fabian Monrose, and Gregory Blanc, editors, *Research in Attacks, Intrusions, and Defenses*, volume 9404, pages 317–338, Cham, 2015. Springer International Publishing. https://doi.org/10.1007/978-3-319-26362-5_15.

[48] Yufan You, Jianjun Chen, Qi Wang, and Haixin Duan. Artifacts for "My ZIP isn't your ZIP: Identifying and Exploiting Semantic Gaps Between ZIP Parsers" (USENIX Security '25), May 2025. https://doi.org/10.5281/zenodo.15526863.

[49] Jiahe Zhang, Jianjun Chen, Qi Wang, Hangyu Zhang, Chuhan Wang, Jianwei Zhuge, and Haixin Duan. Inbox Invasion: Exploiting MIME Ambiguities to Evade Email Attachment Detectors. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 467–481, Salt Lake City UT USA, December 2024. ACM. https://doi.org/10.1145/3658644.3670386.

[50] Linkai Zheng, Xiang Li, Chuhan Wang, Run Guo, Haixin Duan, Jianjun Chen, Chao Zhang, and Kaiwen Shen. ReqsMiner: Automated Discovery of CDN Forwarding Request Inconsistencies and DoS Attacks with Grammar-based Fuzzing. In *Proceedings 2024 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2024. Internet Society. https://doi.org/10.14722/ndss.2024.24031.

[51] Thierry Zoller. Anti-Virus archive bypasses explained, April 2009. https://blog.zoller.lu/2009/04/case-for-av-bypassesevasions.html.

[52] Thierry Zoller. Advisories 2020, 2020. https://blog.zoller.lu/2020/01/a-new-blog-post-since-last-one-in-2013.html.

A Algorithms

Pseudocode for the algorithms used in Section 4 are listed in Algorithm 1, Algorithm 2, and Algorithm 3.

Algorithm 1 Directory Hash Computation

```
Input: The path to the directory.
Output: The hash value of the directory Hash(path).
1: $H \leftarrow$ empty hasher
2: if path is a symbolic link then
3:     Update $H$ with 'L'
4:     Update $H$ with the link target
5: else if path is a regular file then
6:     Update $H$ with 'F'
7:     Update $H$ with the file content
8: else if path is a directory then
9:     Update $H$ with 'D'
10:    $C \leftarrow$ empty list
11:    for each entry in the directory do
12:        $digest \leftarrow Hash(entry)$
13:        if digest is empty then
14:            Skip to the next entry            // empty directory
15:        end if
16:        $H_e \leftarrow$ empty hasher
17:        if entry has special characters in base name then
18:            // Ignore inconsistent special characters
19:            Update $H_e$ with 'S'
20:        else
21:            Update $H_e$ with 'N'
22:            Update $H_e$ with the base name of entry
23:        end if
24:        Update $H_e$ with digest
25:        Insert $H_e.finalize()$ into $C$
26:    end for
27:    if $C$ is empty then
28:        return empty
29:    end if
30:    Sort $C$ alphabetically    // entry order does not matter
31:    for each digest in $C$ do
32:        Update $H$ with digest
33:    end for
34: end if
35: return $H.finalize()$
```

Algorithm 2 Interesting Sample Detection

```
Input: Parser outputs $O$ corresponding to the input sample.
Output: Whether the sample is interesting. The sample is
    inserted into the corpus if it is interesting. Old samples
    covered by the new sample are removed from the corpus.
 1: $ok \leftarrow \{parser \mid O(parser)$ is not failure$\}$
 2: $incons \leftarrow \{(p,q) \in ok^2 \mid \text{Hash}(O(p)) \neq \text{Hash}(O(q))\}$
 3: for each $s$ in the corpus do
 4:     if $ok \subseteq s.ok$ and $incons \subseteq s.incons$ then
 5:         return $false$
 6:     end if
 7:     if $ok \supseteq s.ok$ and $incons \supseteq s.incons$ then
 8:         Remove $s$ from the corpus
 9:     end if
10: end for
11: Insert the sample and $(ok, incons)$ into the corpus
12: return $true$
```

Algorithm 3 Fuzzing With UCB-Based Mutation Selection

```
Input: $K$ mutation strategies, batch size $B$, UCB weight de-
    caying rate $\alpha$, softmax temperature $\beta$.
Output: A corpus of interesting ZIP file samples.
 1: Initialize $R$ and $N$ as arrays of $K$ zeros // reward & count
 2: loop
 3:     for $i \leftarrow 1$ to $K$ do
 4:         $R(i) \leftarrow \alpha \cdot R(i)$                // decay weights
 5:         $N(i) \leftarrow \alpha \cdot N(i)$
 6:     end for
 7:     $n \leftarrow \max\left(\sum_{i=1}^{K} N(i), 1\right)$     // $\max(\cdot,1)$ to avoid $\ln 0$
 8:     $w_i \leftarrow \frac{R(i)}{\max(N(i),1)} + \sqrt{\frac{2\ln n}{\max(N(i),1)}}$     // avoid div by 0
 9:     $\sigma(\mathbf{w})_i \leftarrow \frac{e^{\beta w_i}}{\sum_{j=1}^{K} e^{\beta w_j}}$      // apply softmax to UCB
10:     $I \leftarrow$ empty list
11:     for $i \leftarrow 1$ to $B$ do
12:         Select a seed $s$ from the corpus
13:         $M \leftarrow$ empty list
14:         loop
15:             $m \leftarrow$ a random mutation with weights $\sigma(\mathbf{w})$
16:             $s \leftarrow$ mutate $s$ by strategy $m$
17:             Insert $m$ into $M$
18:             Break loop with 50% probability
19:         end loop
20:         Insert $(s, M)$ into $I$
21:     end for
22:     Test all inputs $I$ against every parser in parallel
23:     for each $(s, M)$ in $I$ do
24:         for each $m$ in $M$ do
25:             if sample $s$ is detected as interesting then
26:                 $R(m) \leftarrow R(m) + \frac{1}{|M|}$
27:             end if
28:             $N(m) \leftarrow N(m) + \frac{1}{|M|}$
29:         end for
30:     end for
31: end loop
```

B Tested Parsers

We tested the ZIP parsers listed in Table 3. Asterisks are used to mark parsers that are provided as a built-in feature of the programming language, such as standard libraries.

C Parser Inconsistency Table

Table 4 provides the number of inconsistency types between ZIP parser pairs. The numbers could be inaccurate when comparing a standard mode parser with a streaming mode parser, because test samples for other ambiguity types, especially those in the ZIP structure positioning category, might cause inconsistencies due to different parsing modes. The complete list of inconsistency types can be found in the artifacts.

Table 3: **Tested ZIP parsers.** GitHub stargazer counts were retrieved on June 12, 2025.

#	Name (API)	Language (*built-in)	Version	GitHub Star
1	Info-ZIP	C	6.0	-
2	7-Zip	C++	24.08	1.5k
3	p7zip	C++	16.02	-
4	WinRAR	C++	7.01	-
5	Zip-Ada	Ada	59	28
6	go-unarr	C	0.2.4	292
7	libarchive	C	3.7.7	3.2k
8	libzip	C	1.10.1	918
9	minizip	C	1.3.1	6.2k
10	minizip-ng	C	4.0.8	1.3k
11	zip	C	0.3.2	1.5k
12	zziplib	C	0.13.78	68
13	DotNetZip	C#	1.16.0	550
14	SharpCompress	C#	0.38.0	2.4k
15	SharpZipLib	C#	1.4.2	3.8k
16	System.IO.Compression	C#*	9.0.0	-
17	Android libziparchive	C++	34.0.5	-
18	POCO	C++	1.13.3	9k
19	std.zip	D*	2.109.1	-
20	archive	Dart	3.6.1	445
21	zip	Erlang*	27.1.2.0	-
22	archive/zip	Go*	1.22.3	-
23	zip	Haskell	2.1.0	84
24	zip-archive	Haskell	0.4.3.2	46
25	Commons Compress (stream)	Java	1.27.1	365
26	Commons Compress (ZipFile)	Java	1.27.1	365
27	java.util.zip.ZipFile	Java*	21.0.5	-
28	java.util.zip.ZipInputStream	Java*	21.0.5	-
29	zip4j (ZipFile)	Java	2.11.5	2.2k
30	zip4j (ZipInputStream)	Java	2.11.5	2.2k
31	@ronomon/zip	JavaScript	1.12.0	262
32	adm-zip	JavaScript	0.5.16	2.1k
33	decompress-zip	JavaScript	0.3.3	102
34	jszip	JavaScript	3.10.1	10k
35	node-stream-zip	JavaScript	1.15.0	462
36	unzipper (Extract)	JavaScript	0.12.3	458
37	unzipper (Open)	JavaScript	0.12.3	458
38	yauzl	JavaScript	2.10.0	766
39	yauzl	JavaScript	3.2.0	766
40	zip.js	JavaScript	2.7.53	3.6k
41	PharData	PHP*	8.3.13	-
42	phpzip	PHP	4.0.2	495
43	paszlib	Pascal*	3.2.2	-
44	Archive::Zip	Perl	1.68	16
45	zipfile	Python*	3.13.0	-
46	file/unzip	Racket*	8.15	-
47	rubyzip (File)	Ruby	2.3.2	1.4k
48	rubyzip (InputStream)	Ruby	2.3.2	1.4k
49	zip	Rust	2.2.0	725
50	ZIP Foundation	Swift	0.9.19	2.5k

Table 4: **Number of inconsistency types between ZIP parser pairs.** The row and column headers correspond to the parser numbers in Table 3. The internal cells are the numbers of inconsistency types between parser pairs.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
1	-	5	6	6	4	6	5	6	7	4	7	6	3	6	5	4	4	7	6	6	4	4	8	6	5	5	4	5	6	6	1	7	5	5	5	7	5	6	7	6	7	6	7	6	4	5	5	6	4	5
2	5	-	2	8	6	8	7	7	9	6	8	9	7	6	8	7	6	4	6	6	7	6	8	5	5	7	4	3	8	6	1	10	7	7	8	7	8	8	8	7	9	9	8	7	7	4	8	6	7	6
3	6	2	-	7	6	8	7	8	7	7	8	8	7	6	9	7	7	4	6	7	7	5	9	5	6	8	5	4	9	5	1	10	7	6	7	7	8	7	8	8	8	8	7	6	5	4	8	6	7	6
4	6	8	7	-	8	10	8	10	8	8	9	10	8	11	7	9	9	10	10	11	10	9	11	10	10	8	8	11	9	10	3	9	9	8	8	10	10	9	9	10	11	9	11	9	7	11	9	10	9	9
5	4	6	6	8	-	11	8	11	7	8	9	8	8	9	5	8	6	6	9	10	7	7	13	6	9	8	6	7	8	6	2	8	9	8	7	7	9	10	10	9	10	8	7	9	6	6	7	8	7	8
6	6	8	8	10	11	-	8	10	10	6	5	9	6	12	7	8	7	11	8	8	9	8	11	8	12	9	9	12	4	12	2	8	9	9	8	12	7	5	5	9	8	7	10	10	9	12	8	10	9	8
7	5	7	7	8	8	8	-	9	7	4	6	9	7	9	6	6	8	8	6	6	9	7	10	8	9	5	4	9	7	9	1	9	8	5	6	10	8	5	5	8	10	7	10	9	5	9	8	8	8	9
8	6	7	8	10	11	10	9	-	11	10	7	9	8	10	8	8	8	10	5	9	8	7	7	7	11	10	7	9	9	10	2	9	9	9	9	10	8	8	6	9	8	6	9	10	8	9	9	10	8	8
9	7	9	7	8	7	10	7	11	-	8	9	9	7	11	4	9	7	9	8	10	9	9	12	7	10	8	7	10	7	9	2	9	9	9	8	10	9	10	10	10	10	9	10	9	8	10	8	10	9	9
10	4	6	7	8	8	6	4	10	8	-	4	6	5	9	5	5	6	8	7	7	8	4	8	7	8	5	3	8	4	8	2	7	6	6	6	9	6	4	4	7	7	7	10	7	6	8	8	8	7	7
11	7	8	8	9	9	5	6	7	9	4	-	6	5	11	4	4	5	9	5	7	7	6	7	8	10	7	6	10	3	10	2	6	6	6	6	10	5	4	3	5	7	5	8	7	7	10	7	10	6	5
12	6	9	8	10	8	9	9	9	9	6	6	-	4	10	6	5	3	7	7	10	8	6	9	6	11	9	5	9	7	10	2	5	6	9	5	10	6	7	8	6	3	5	9	8	7	8	8	10	6	5
13	3	7	7	8	8	6	7	8	7	5	5	4	-	7	5	3	5	5	6	7	6	4	7	5	9	8	3	6	6	8	1	6	4	6	4	7	5	6	6	6	4	6	7	6	6	6	6	7	7	4
14	6	6	6	11	9	12	9	10	11	9	11	10	7	-	8	7	8	5	10	11	8	9	12	2	8	11	7	5	10	7	1	12	8	8	7	4	8	11	11	10	9	10	9	11	9	5	9	7	10	9
15	5	8	9	7	5	7	6	8	4	5	4	6	5	8	-	4	6	6	5	6	7	5	7	5	9	5	3	7	7	7	1	6	6	7	3	7	4	8	7	7	6	5	5	5	6	6	7	8	4	5
16	4	7	7	9	8	8	6	8	9	5	4	5	3	7	4	-	3	7	6	9	6	4	8	4	10	8	3	7	5	9	1	6	4	5	4	9	1	4	5	5	4	5	8	6	6	8	6	9	6	5
17	4	6	7	9	6	7	8	8	7	6	5	3	5	8	6	3	-	7	3	8	3	3	7	6	9	8	5	6	4	7	1	6	4	6	5	8	3	5	5	6	3	3	5	7	6	7	4	8	6	2
18	7	4	4	10	6	11	8	10	9	8	9	7	5	5	6	7	7	-	8	11	6	6	12	4	5	10	7	2	8	2	2	9	8	8	8	4	8	10	10	9	7	8	6	8	6	2	7	3	7	7
19	6	6	6	10	9	8	6	5	8	7	5	7	6	10	5	6	3	8	-	5	4	5	5	5	10	8	4	8	5	9	2	8	7	9	6	9	6	7	5	9	4	5	6	8	6	8	6	10	7	5
20	6	6	7	11	10	8	6	9	10	7	7	10	7	11	6	9	8	11	5	-	10	9	9	9	10	9	8	11	5	11	0	9	8	10	5	11	9	8	7	10	11	10	10	10	10	12	8	10	10	8
21	4	7	7	10	7	9	9	8	9	8	7	8	6	8	7	6	3	6	4	10	-	7	9	6	10	10	8	7	6	8	1	9	9	10	6	8	7	7	7	8	6	5	6	10	7	6	5	8	6	5
22	4	6	5	9	7	8	7	7	9	4	6	6	4	9	5	4	3	6	5	9	7	-	8	5	9	8	3	7	6	7	2	7	6	6	7	10	5	5	6	7	5	6	7	5	6	7	8	10	3	3
23	8	8	9	11	13	11	10	7	12	8	7	9	7	12	7	8	7	12	5	9	9	8	-	8	13	9	6	12	9	13	3	9	8	9	10	12	7	8	7	9	7	7	12	10	8	12	11	12	8	8
24	6	5	5	10	6	8	8	7	7	7	8	6	5	2	5	4	6	4	5	9	6	5	8	-	5	9	4	3	8	4	1	9	5	6	5	5	4	9	9	7	5	7	7	6	6	2	6	5	7	5
25	5	5	6	10	9	12	9	11	10	8	10	11	9	8	9	10	9	5	10	10	10	9	13	5	-	9	10	4	9	3	1	10	10	10	8	7	11	10	10	11	11	10	10	12	9	5	9	3	9	11
26	5	7	8	8	8	9	5	10	8	5	7	9	8	11	5	8	8	10	8	9	10	8	9	9	9	-	5	11	7	10	1	8	8	6	8	9	8	7	6	8	10	7	11	8	6	11	10	9	7	10
27	4	4	5	8	6	9	4	7	7	3	6	5	3	7	3	3	5	7	4	8	8	3	6	4	10	5	-	7	7	8	0	8	3	4	6	10	4	7	7	7	5	5	8	4	3	7	8	9	5	5
28	5	3	4	11	7	12	9	9	10	8	10	9	6	5	7	7	6	2	8	11	7	7	12	3	4	11	7	-	9	1	0	11	8	9	7	7	8	11	11	10	8	8	7	10	6	1	8	2	8	8
29	6	8	9	9	8	4	7	9	7	4	3	7	6	10	7	5	4	8	5	5	6	6	9	8	9	7	7	9	-	9	2	7	6	8	5	8	5	4	2	7	7	5	6	8	8	9	5	8	8	5
30	6	6	5	10	6	12	9	10	9	8	10	10	8	7	7	9	7	2	9	11	8	7	13	4	3	10	8	1	9	-	2	10	9	10	8	7	10	11	11	11	10	9	7	10	8	2	8	2	7	9
31	1	1	1	3	2	2	1	2	2	2	2	2	1	1	1	1	1	2	2	0	1	2	3	1	1	1	0	0	2	2	-	3	1	1	1	2	1	2	2	0	2	2	3	1	1	1	1	1	1	1
32	7	10	10	9	8	8	9	9	9	7	6	5	6	12	6	6	6	9	8	9	9	7	9	9	10	8	8	11	7	10	3	-	7	8	5	9	7	7	7	7	6	6	10	8	9	11	9	11	7	6
33	5	7	7	9	9	9	8	9	9	6	6	6	4	8	6	4	4	8	7	8	9	6	8	5	10	8	3	8	6	9	1	7	-	6	6	10	4	7	7	6	6	7	9	7	7	9	8	10	7	6
34	5	7	6	8	8	9	5	9	9	6	6	9	6	8	7	5	6	8	9	10	10	6	9	6	10	6	4	9	8	10	1	8	6	-	8	11	7	7	7	7	10	7	10	7	5	9	10	10	7	8
35	5	8	7	8	7	8	6	9	8	6	6	5	4	7	3	4	5	8	6	5	6	7	10	5	8	8	6	7	5	8	1	5	6	8	-	7	5	8	8	7	7	6	7	9	7	7	5	8	6	5
36	7	7	7	10	7	12	10	10	10	9	10	10	7	4	7	9	8	4	9	11	8	10	12	5	7	9	10	7	8	7	2	9	10	11	7	-	9	10	10	11	10	9	9	11	9	6	10	7	9	10
37	5	8	8	10	9	7	8	8	9	6	5	6	5	8	4	1	3	8	6	9	7	5	7	4	11	8	4	8	5	10	1	7	4	7	5	9	-	5	6	7	5	5	7	5	7	8	7	10	6	4
38	6	8	7	9	10	5	5	8	10	4	4	7	6	11	8	4	5	10	7	8	7	5	8	9	10	7	7	11	4	11	2	7	7	7	8	10	5	-	1	7	6	5	8	8	7	11	8	9	6	7
39	7	8	8	9	10	5	5	6	10	4	3	8	6	11	7	5	5	10	5	7	7	6	7	9	10	6	7	11	2	11	2	7	7	7	8	10	6	1	-	6	7	4	7	7	7	11	8	9	6	6
40	6	7	8	10	9	9	8	9	10	7	5	6	6	10	7	5	6	9	9	10	8	7	9	7	11	8	7	10	7	11	0	7	6	7	7	11	7	7	6	-	8	6	9	8	8	10	8	11	6	7
41	7	9	8	11	10	8	10	8	10	7	7	3	4	9	6	4	3	7	4	11	6	5	7	5	11	10	5	8	7	10	2	6	6	10	7	10	5	6	7	8	-	6	9	7	8	8	8	10	5	3
42	6	9	8	9	8	7	7	6	9	7	5	5	6	10	5	5	3	8	5	10	5	6	7	7	10	7	5	8	5	9	2	6	7	7	6	9	5	5	4	6	6	-	7	6	5	8	7	10	6	4
43	7	8	7	11	7	10	10	9	10	10	8	9	7	9	5	8	5	6	6	10	6	7	12	7	10	11	8	7	6	7	3	10	9	10	7	9	7	8	7	9	9	7	-	9	10	7	7	9	6	5
44	6	7	6	9	9	10	9	10	9	7	7	8	6	11	5	6	7	8	8	10	10	5	10	6	12	8	4	10	8	10	1	8	7	7	9	11	5	8	7	8	7	6	9	-	5	10	9	11	6	6
45	4	7	5	7	6	9	5	8	8	6	7	7	6	9	6	6	6	6	6	10	7	6	8	6	9	6	3	6	8	8	1	9	7	5	7	9	7	7	7	8	8	5	10	5	-	6	9	9	8	6
46	5	4	4	11	6	12	9	9	10	8	10	8	6	5	6	8	7	2	8	12	6	7	12	2	5	11	7	1	9	2	1	11	9	9	7	6	8	11	11	10	8	8	7	10	6	-	8	3	8	8
47	5	8	8	9	7	8	8	9	8	8	7	8	6	9	7	6	4	7	6	8	5	8	11	6	9	10	8	8	5	8	1	9	8	10	5	10	7	8	8	8	8	7	7	9	9	8	-	10	8	4
48	6	6	6	10	8	10	8	10	10	8	10	10	7	7	8	9	8	3	10	10	8	10	12	5	3	9	9	2	8	2	1	11	10	10	8	7	10	9	9	11	10	10	9	11	9	3	10	-	10	10
49	4	7	7	9	7	9	8	8	9	7	6	6	7	10	4	6	6	7	7	10	6	3	8	7	9	7	5	8	8	7	1	7	7	7	6	9	6	6	6	6	5	6	6	6	8	8	8	10	-	5
50	5	6	6	9	8	8	9	8	9	7	5	5	4	9	5	5	2	7	5	8	5	3	8	5	11	10	5	8	5	9	1	6	6	8	5	10	4	7	6	7	3	4	5	6	6	8	4	10	5	-