Email Smuggling with Differential Fuzzing of MIME Parsers

Seyed Behnam Andarzian
ICIS, Radboud University
seyedbehnam.andarzian@ru.nl

Martin Meyers
ICIS, Radboud University
martin.meyers@ru.nl

Erik Poll
ICIS, Radboud University
erikpoll@cs.ru.nl

***Abstract*—A single email gets parsed multiple times: by the mail server receiving the email, by virus or spam filters, and finally by the mail client that displays the email to the user. MIME (Multipurpose Internet Mail Extensions) is a standard that extends the format of email messages, allowing them to include multimedia content, attachments, and non-ASCII text.**

Ensuring that email content is correctly formatted and interpreted across different systems is crucial. Differentials in how applications parse the same MIME message can have security implications. For instance, a virus or spam filter might ignore part of the data that ends up being processed by the mail client.

We present experiments with differential fuzzing to discover differentials in how MIME parsers handle the same message. We investigate the root causes and see if the differentials can be exploited. Our research reveals many parser differentials in MIME parsers, including some that can be exploited to smuggle emails past virus and spam filters. On top of that, our experiments found many memory corruption bugs.

I. Introduction

When email was introduced initially, security and safety were not taken into account[1]. Over the years, this has resulted in lots of malicious email traffic, such as spam and emails spreading viruses. Filters were added to block spam and viruses. Also, protocols such as SPF, DKIM, and DMARC were added to prevent spoofing of the sender address.

Spam and virus filters are often separate systems from the mail clients. This means that incoming emails are parsed several times by different parsers. If these parsers behave differently, a spam or virus filter might ignore parts of emails.

Differentials in how systems parse data (a.k.a. parser differentials) occur when different systems (such as virus filters, spam filters, or mail clients) interpret the same email message differently. These differentials can create security gaps that allow malicious content to slip through undetected by one component but processed by another. These differentials will be quickly detected and repaired if they affect functionality. However, differentials that only manifest themselves in unusual circumstances are unlikely to be noticed and fixed, potentially leading to security problems later. Security problems caused by parser differentials include HTTP response splitting [2] and ambiguities in X.509 certificates [3].

Fuzzing is generally used to find program vulnerabilities. It is typically performed by sending inputs to a program and checking whether it crashes or hangs. In differential fuzzing, the results of two or more implementations are compared to spot differentials. As discussed above, this can uncover security issues arising from parser differentials.

This paper presents the following contributions:

- We show that differential fuzzing can be used to find (many!) parser differentials between MIME parsers.
- To do this, we provide a harness for the differential fuzzing of MIME parsers.
- We analyze the exploitability of the parser differentials.

Sections II and III provide the necessary background for our research and Section IV discusses the systems we analyse. After that, Section V discusses the details of our differential fuzzing setup. Section VI discusses the parser differentials discovered. We test their exploitability in section VII. Section VIII discusses related work and section IX options to improve our work. Finally, section X summarizes our main conclusions.

II. Email protocols

When an email is sent, several steps and protocols work together to ensure the message is properly formatted, transmitted, and delivered. These steps include:

- Sending the email: After composing an email in an application (such as Gmail or Outlook), the email client uses the SMTP (Simple Mail Transfer Protocol) to transmit the message to the recipient's mail server.
- Processing by the recipient's server: Once the email reaches the recipient's server, it undergoes various checks, such as spam and virus filtering, before it is delivered to the recipient's mailbox.
- Viewing the email: The recipient's mail client (such as Thunderbird or Outlook) retrieves the email and presents it to the user, displaying the content, attachments, or HTML format as needed.

Behind the scenes, several standards ensure that emails are formatted correctly, transmitted safely, and presented accurately. These protocols include SMTP for transmission and IMF and MIME for structuring and encoding the email content.

A. SMTP

SMTP is the standard protocol for sending emails from one server to another over the internet. When an email is sent,

[1]There is no mention of security in the SMTP RFCs until RFC 2821 from 2001 [1].

SMTP establishes communication between the sender's and recipient's email servers. Several commands and responses are exchanged during this process and transmit the email.

SMTP does not dictate how an email is formatted. It does requires the headers to follow the IMF format and the body to adhere to the MIME format, especially when dealing with non-ASCII content or attachments.

B. IMF (Internet Message Format)

IMF defines the standard structure of an email. It ensures that emails are organized so any email client can interpret them. Important aspects for our research are:

- Headers: IMF mandates specific headers such as `From`, `To`, `Subject`, and `Date`. These headers provide key information about the email and must follow the structure `<HEADER NAME> : <HEADER BODY>`.
- Line Wrapping: To prevent overly long lines in headers, IMF allows for line wrapping, where a line can be split by ending with a Carriage Return and Line Feed (a.k.a CRLF or `\r\n`) and continuing on the next line with leading whitespace. So for example the header

  ```
  Subject: This
   is a test
  ```

  that contains a CRLF should be treated just like

  ```
  Subject: This is a test
  ```

  according to IMF.

IMF handles the basic structure of an email but doesn't account for more complex content such as attachments or non-ASCII characters. For that, we turn to MIME.

C. MIME (Multipurpose Internet Mail Extensions)

MIME expands the capabilities of IMF by allowing emails to include a wide range of content types, such as attachments, multimedia, and non-ASCII text. It introduces several key elements that extend basic email functionality:

1) MIME Headers: MIME introduces additional headers that specify the type and encoding of the content in the email. These headers, that are critical in determining how the email body is processed and displayed, are:

- `Content-Type`: This header specifies the type of content contained in the email (e.g., `text/plain` for plain text, `text/html` for HTML emails, or `multipart/*` for emails containing multiple parts, such as attachments).
- `Content-Transfer-Encoding`: This header defines the encoding used for the email content, which is especially important when transmitting non-ASCII or binary data over systems that only support ASCII. It ensures that the data can be correctly interpreted when received.

2) Multipart Messages: One of MIME's important features is its ability to handle multipart messages. A multipart message allows the email to be divided into separate parts, each of which can contain different types of content. For example, an email may contain plain text, an HTML version, and attachments, all in a single message. Different types of multipart messages are:

- Multipart/Alternative: This is commonly used to send both a plain text version and an HTML version of an email. The recipient's email client chooses which version to display based on user preferences.
- Multipart/Mixed: This type is used for emails with attachments. Each part of the message is separated by a boundary string that indicates where one part ends and another begins.
- Multipart-Preamble: MIME also includes a part called the multipart-preamble, which is used to include data that is not meant to be displayed by the recipient's email client. This is typically ignored but can be used for special cases.

3) Content-Transfer-Encoding: Since SMTP only supports US-ASCII (7-bit) characters, MIME provides encoding options that allow the inclusion of non-ASCII or binary data in emails. There are two common encoding methods:

- Base64 encoding: This encoding allows binary data (such as attachments or images) to be converted into ASCII text for transmission. base64-encoded content can be safely transmitted over SMTP and decoded by the recipient's mail client.
- Quoted-Printable Encoding: This encoding is used for text that contains mostly ASCII characters but includes a few non-ASCII characters. It allows characters to be encoded in a human-readable format, with non-ASCII characters encoded as their hexadecimal representation. For example, the letter `A` might appear as `=41`, which becomes the character `"A"` after decoding.

Because different mail servers and clients may interpret encoded data differently, properly handling these MIME features is critical. For example, base64 encoding uses a specific alphabet (a-z, A-Z, 0-9, + and /); any character outside this alphabet should be ignored. Some parsers, however, may not follow this rule strictly, leading to differentials.

D. Common features of SMTP, IMF, and MIME

SMTP, IMF, and MIME all rely on Carriage Return and Line Feed (a.k.a CRLF or `\r\n`) to indicate the end of a line. This ensures consistent interpretation of email content across different platforms and email clients. Carriage Return and Line Feed are ASCII characters and SMTP specification effectively requires messages to be ASCII-only.

III. Data Smuggling

Data smuggling refers to bypassing security mechanisms, such as filters or firewalls, to deliver data that should be blocked. The technique can be applied in various contexts, ranging from web requests to email communications, and it often exploits differentials in how systems parse and interpret the same data.

Several types of data smuggling exist, for instance:

- HTTP Request Smuggling: This involves sending specially crafted HTTP requests that exploit differentials

in how web servers, proxies or load balancers interpret and handle requests. By smuggling data through these intermediaries, an attacker may be able to bypass security measures or cause systems to behave unexpectedly [4].
- HTML Smuggling: This technique involves using JavaScript to smuggle malicious content, such as malware, into a web browser. The malicious content in HTML smuggling involves using JavaScript to reconstruct or decode a hidden or obfuscated payload—often a binary file like an executable, a ZIP archive, or other malicious file types—within the victim's browser, bypassing traditional perimeter defenses like firewalls and email filters [5].

In this paper we look at email smuggling. Here attackers try to craft email messages to pass through virus and spam filters that should block them. By exploiting weaknesses in how different email system components handle these messages, attackers can inject malicious content into a mail client. Email smuggling becomes possible when parser differentials can be exploited in a full email setup (see Figure 1), meaning the email message passes through multiple stages—including the SMTP handler and potentially several filters—while remaining undetected. If the SMTP handler or a filter rejects the email due to the differential, the attack fails because the message never reaches the vulnerable MIME parsers. However, if the email successfully arrives at the mail client with its malicious content intact, the differential can be exploited, effectively bypassing the intended security controls.

Smuggling techniques exploit differentials in how systems parse or process incoming data. In the case of email smuggling and this paper, the attack leverages parser differentials between parsers for the MIME format.

IV. Targets

The targets in our research are the different kinds of components in the email ecosystem that parse emails: mail servers, spam filters, virus scanners and mail clients. As explained earlier, each of these have their own role in the email processing workflow:

- Mail servers are responsible for receiving emails and applying initial filters, including internal spam and virus detection mechanisms.
- Spam and virus filters inspect the email's content, ensuring that no malicious or unwanted content passes through. If these filters do not correctly parse the email's MIME structure, this may result in false negatives where the filter fail to spot some malicious or unwanted content.
- Mail clients are the final step, displaying the email to the user. They must correctly interpret the parsed MIME data to render the email content, attachments, and any other included elements properly.

We selected open-source targets to analyze, allowing us to extract the parser code we wanted to analyze from any surrounding code. For the C(++) implementations it also allowed us to instrument the code for grey-box fuzzing.

Table I provides an overview of the selected targets. We briefly describe them below.

Postfix is an open-source mail server that manages the receipt and routing of incoming emails. After receiving an email, Postfix processes it through its spam and virus filters before it is accessible by mail clients. It performs checks on the MIME structure, such as validating the content type and ensuring the multipart boundaries conform to standards (as discussed in section II). Emails with invalid MIME headers or formats are rejected at this stage.

SpamAssassin is an open-source spam filter that inspects incoming emails to determine whether they are spam. It applies a wide range of rules and heuristics to analyze the email's content. Parsing the MIME structure correctly is essential for SpamAssassin, as improperly parsed data might result in false negatives, allowing spam messages to reach the inbox.

ClamAV is an open-source virus filter that scans incoming emails for malware or viruses. It parses the MIME data to inspect attachments or other encoded content that could contain malicious payloads. Correct MIME parsing is vital for ClamAV to detect and block harmful emails.

Evolution is an open-source email client that users interact with to read and manage their emails. It is responsible for rendering the MIME-encoded content into a readable format, including displaying text, HTML, and attachments. Accurate parsing is necessary for the client to correctly display the email content and handle multipart messages.

Project	Version	Language	Function
Postfix	3.8.2	C	Mail server
SpamAssassin	4.0.0	Perl	Spam filter
ClamAV	1.2.0	C	Virus filter
Evolution	3.50.0	C	Mail client

TABLE I: Differential fuzzing targets

V. Differential Fuzzing Set-up

Our setup for identifying parser differentials between MIME parsers consists of two main components:

1) A fuzzer to generate MIME messages that are sent to the targets.
2) A test oracle to evaluate whether the targets parse a message differently.

Figure 2 gives an overview of the setup. The most challenging part of this setup is the test oracle. Typically, fuzzers check if a target crashes or hangs when processing an input, often aided by instrumentation that detects misbehavior such as memory corruption. However, determining if multiple targets parse the same input differently is more complex.

A. Fuzzer

In our experiments we used two fuzzers: AFL++ [6], a grey-box fuzzer, and T-Reqs, a grammar-based fuzzer[2]. These

[2]The experiments using T-Reqs were initially carried out as a Master thesis project [7].

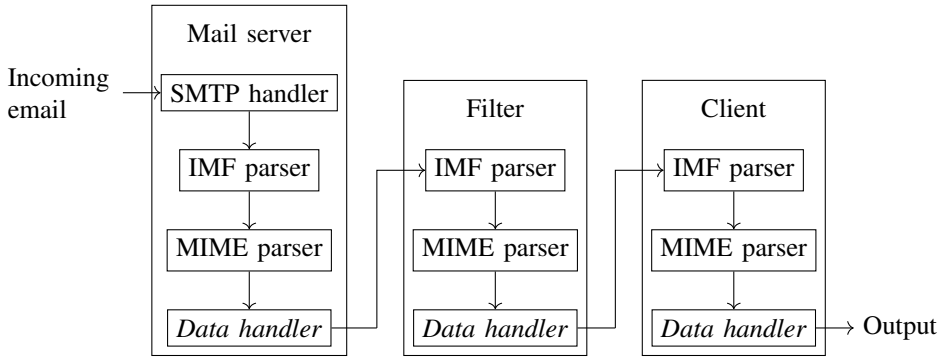

Fig. 1: Email handling by a full email setup with a single filter; this filter could be a spam filter or an antivirus solution.

different types of fuzzers turned out to have different strengths, as we will discuss later: AFL++ was better at uncovering memory corruption bugs, while T-Reqs was better at in identifying parser differentials. But even when using T-Reqs we still needed AFL++'s test corpus minimization tool to reduce the number of test cases found to a manageable number, discussed in more detail below.

The input grammar of T-reqs and the initial seeds for AFL++ were crafted/chosen to cover a variety of MIME features and cover the same features, to aim for good and comparable coverage between experiments. The seeds included:

- MIME headers such as `Content-Type` with different values (e.g., multipart, plain text, HTML),
- `Content-Transfer-Encoding` headers, using various encoding schemes (e.g., 7-bit, 8-bit, and base64),
- email bodies with variations such as plain text, base64-encoded text, and multipart data.

1) Grey-Box Fuzzing with AFL++: AFL++ is a grey-box fuzzer that uses instrumentation to gather code coverage feedback from the target applications. This feedback guides the fuzzer in generating test cases that thoroughly exercise the target's code paths.

The test cases generated by AFL++ are sent to the targets through a harness that forwards them to the four different MIME parsers and compares the results. In our set-up AFL++ treats the entire test harness, including the target systems implemented in C (i.e. Postfix, ClamAV, and Evolution), as a single target. So the code coverage feedback encompasses the aggregated execution paths through all these instrumented components. SpamAssassin, being written in Perl, was excluded from AFL++ instrumentation. We also excluded the test harness comparison operations from AFL++ instrumentation to prevent them from influencing test case generation.

2) Grammar-Based Fuzzing with T-Reqs: T-Reqs is a black-box grammar-based fuzzer. Unlike a grey-box fuzzer like AFL++ it requires a grammar of the input format that is the basis for generating inputs, so we had to provide T-Reqs with a MIME grammar.

A downside of a black-box fuzzer like T-Reqs compared to a greybox fuzzer like AFL++ is that if it finds a bugs (or in our case, a differential) it tends to find many instances of essentially the same bug (or differential). A greybox fuzzer automatically weeds out such duplicate findings: the code coverage feedback is not just useful to spot new execution paths, but also to spot if two test cases generate the same execution path. Indeed, in our experiments for every differential that T-Reqs found it produced many test cases that triggered it. We used AFL++'s `afl-cmin` corpus minimization tool to weed out these duplicates.

3) Insights from Using Both Fuzzers: The combination of AFL++ and T-Reqs yielded complementary results, underscoring the importance of employing multiple fuzzing approaches for a comprehensive evaluation. Specifically:

- AFL++: As a coverage-guided fuzzer, AFL++ excelled in finding memory corruption bugs, which were prevalent in the C-based targets (Postfix, ClamAV, and Evolution).
- T-Reqs: By leveraging its understanding of MIME grammar, T-Reqs was able to uncover more parser differentials between the targets. These differentials typically reflected differenes in how MIME structures were interpreted, some of which AFL++ could not find due to its lack of semantic understanding of the MIME grammar.

For example, while AFL++ found numerous memory corruption bugs in Postfix and Evolution, it struggled to detect subtle differentials in how MIME headers like `Content-Type` were parsed. On the other hand, T-Reqs, with its grammar-based approach, exposed several parser differentials, revealing areas where the same MIME message was handled differently by the two parsers. These differentials, if exploited, could lead to security vulnerabilities such as message misinterpretation or unintended processing behavior.

Figure 2 illustrates the overall fuzzer and harness setup. The grey area represents the instrumented targets for grey-box fuzzing. The flexibility of our harness allowed us to easily switch between AFL++ and T-Reqs.

B. Test Oracle

To evaluate parser differentials, we compared the outputs generated by the targets for the same input MIME messages. Each target typically produces an output (e.g., a parsed MIME message as ASCII text), which serves as a representation of the MIME message. Our test oracle uses Evolution's output

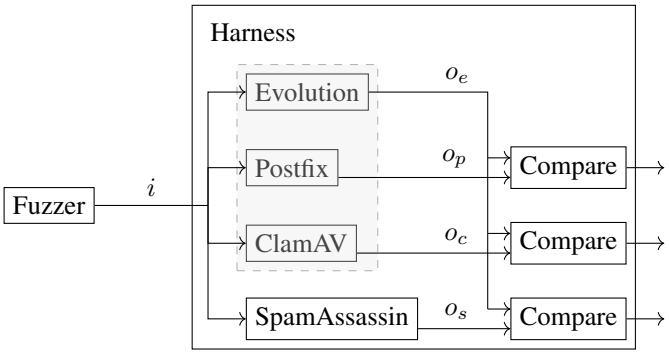

Fig. 2: Fuzzer and harness setup overview. The grey area represents the instrumented targets for grey-box fuzzing.

as the reference and compares it against the outputs of the other targets. So all differentials we find are differences with respect to Evolution. Figure 2 illustrates the data flow through the harness.

There are challenges in comparing outputs directly. For example, some targets produce multiple representations of the same message (e.g., plain text and HTML versions). Additionally, some targets apply formatting changes, such as replacing a carriage return followed by a newline with just a newline; ClamAV and Postfix exhibit this behavior, while Evolution preserves the original formatting. Our test oracle accounts for and ignores such minor formatting differentials to focus on more interesting parser differentials.

If any two targets produce different outputs for the same input, it indicates a possible parser differential. Further analysis is required to determine if these differentials are exploitable and have security implications.

VI. Results

In this section we first present the raw statistics of our results, i.e. how many parser differentials for each of the parsers we found. We then discuss our manual analysis of these differentials, where we tried to look for the interesting ones (that might be exploitable for smuggling email past filters) and we tried to understand the underlying root causes of the differentials.

A. Numbers of Parser Differentials Found

Using AFL++ our differential fuzzing set-up found 448 test cases where there was a parser differential, i.e. for which SpamAssassin, ClamAV or Postfix resulted in a parser differential with Evolution. Using T-Reqs we discovered 349 test cases resulting in such parser differentials. (In fact, the raw output of the T-Reqs fuzzing campaign was many thousands of differentials, but using the corpus minimisation tool `afl-cmin` reduced it to 349 cases.)

These test cases are 'unique' in the sense that they have different execution paths through the combined code of the C targets (i.e. ClamAV, Postfix and Evolution). Since SpamAssassin is written in Perl and cannot be instrumented, the AFL++ fuzzing campaign and AFL++'s corpus minimisation has no insight into SpamAssassin's parsing routines. So there is a chance that the corpus minimisation mistakenly discarded some unique differentials for SpamAssassin, namely if these had an identical execution path for all other targets.

Figures 3 and 4 depict how the parser differentials found by T-Reqs and AFL++ are distributed over SpamAssassin, ClamAV, and Postfix. In these Venn diagrams, each region corresponds to test cases for which one or more parsers diverged in output from Evolution. For example, Figure 3 shows that AFL++ found 3 test cases for which SpamAssassin and ClamAV had a parser differential with Evolution but Postfix had not and 35 test cases for which only Postfix had a parser differential with Evolution. For the overlaps in the Venn diagrams, where several tools showed differential, we cannot tell if these tools have the *same* differential. So for instance, the 42 test cases in the centre of Figure 3 may include cases where Evolution is the outlier and the other three parsers agree with each other, but it may also include cases where all four parsers disagree.

Figure 3 for AFL++ shows large numbers of differentials involving ClamAV and Postfix but very few – in fact , only 1 – that just involves SpamAssassin. This is due to fact that AFL++ is only guided by code coverage of the C code of ClamAV and Postfix while it has no clue about code coverage of SpamAssassin.

Figure 4 shows that T-Reqs finds a more balanced spread of differentials. It finds many more for SpamAssassin, as one would expect given that AFL++'s code coverage guidance is blind here. It also finds more differentials for ClamAV, which is more surprising as here AFL++'s code coverage guidance does work.

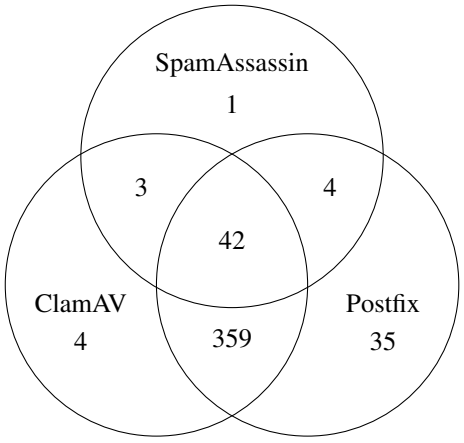

Fig. 3: Distribution of parser differentials found by AFL++ between Evolution and each of the three other parsers: Postfix, ClamAV, SpamAssassin.

B. Analysis of Parser Differentials

We manually analysed the test cases that show differentials to investigate the root cause. The large number made that very labor-intensive, so for the large classes in the Venn diagrams where the fuzzing campaigns produced hundreds of

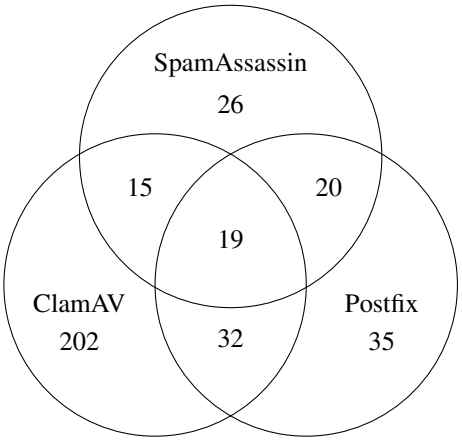

Fig. 4: Distribution of parser differentials found by T-Reqs between Evolution and each of the three other parsers (Postfix, ClamAV, SpamAssassin).

test cases we randomly inspected roughly half and whenever we identified a root cause we tried to weed out similar test cases that involved the same pattern. This means we may have missed some root causes: if these similar test cases that we ignored involved more than one root causes we may have missed the additional ones.) In the end we could identify 15 distinct root causes, listed in Table II. As discussed below in more detail, they fall in roughly two categories, namely those involving malformed or multiple headers and those involving conflicting decodings.

a) Malformed or Multiple Header Handling: These are differentials that stem from the way parsers handle malformed headers (e.g., invalid IMF syntax, extra colons, missing carriage returns) or multiple conflicting headers (e.g., two `Content-Type` or `Content-Transfer-Encoding` fields).

- **D2** (Invalid IMF assumed to be body): If Postfix encounters something it cannot parse as an IMF header, it treats it as the message body. Others ignore it as part of the headers.
- **D3** (*Two* Content-Transfer-Encoding headers): When there are two Content-Transfer-Encoding headers, Evolution decodes the content based on the first header, and SpamAssassin decodes based on the second. ClamAV is aggressive in base64 decoding and decodes anyway, and Postfix does not decode base64.
- **D4** (*Two* Content-Type headers): With two content-type headers, Evolution considers the first, whereas ClamAV and SpamAssassin consider the second. Postfix does not handle the multipart MIME data, so it is unclear how this handles it.
- **D6** (Missing carriage return before body): If a header ends with only `\n` (missing a carriage return before it), and the body does not end with `\r\n`, SpamAssassin does not parse the body at all.
- **D7** (Header line starting with `\r`): ClamAV and Postfix treat such lines as the start of the body, but Evolution does not.
- **D8** (Invalid header that looks like a boundary): Similar to D2, but for SpamAssassin when the invalid IMF header starts with multiple – signs.
- **D10** (Ignoring empty line after invalid header): If there is an invalid header, ClamAV will ignore the next line if it is empty. This means that - where the other parsers will start parsing the body after that - ClamAV continues parsing the data as headers.
- **D11** (Extra colon in `Content-Transfer-Encoding`): When the `Content-Transfer-Encoding` header contains an additional colon (e.g., `Content-Transfer-Encoding:: base64`), SpamAssassin treats the entire header as invalid and ignores it. Evolution, however, parses out the portion before the second colon as a valid header name and continues to interpret the remaining data (e.g., `base64`). It is unclear how ClamAV and Postfix handle this specific scenario, though we observe that ClamAV always attempts base64 decoding and Postfix does not decode base64 at all.
- **D12** (Extra colon before the MIME boundary): When there is an extra colon and data before the boundary in a multipart ContentType header, Evolution will not use the boundary to split the multiple parts. In contrast, ClamAV and SpamAssassin will still parse according to the specified boundary. Postfix splits the message into two parts, which is odd, and we do not know what happens here.
- **D14** (Additional quote in multipart header): SpamAssassin ignores an additional quote in a multipart header while still considering the header data. Evolution ignores the header, ClamAV handles it as an invalid header, and Postfix does not do multipart at all.
- **D15** (Heavily corrupted header/boundary lines): An extreme case of malformed headers and MIME boundaries (e.g., repeated colons, control characters, truncated boundary strings). Postfix treats most invalid lines as body content, ClamAV interprets them as raw data, SpamAssassin retains only minimal header information, and Evolution attempts to "repair" the headers (injecting defaults like `From:`). Because of the widespread corruption and differing "repair" strategies, the parsers produce substantially divergent interpretations.

b) Divergent Decoding Strategies (base64, Quoted-Printable, etc.): Some differentials appear when parsers apply different methods (or strictness) in decoding. For instance, one parser might do partial base64 decoding line-by-line, whereas another decodes the entire body as a single blob.

- **D1** (Postfix does not do base64 decoding): Anything using base64 content transfer encoding will appear differently in Postfix outputs.
- **D5** (differentials in base64 decoding): The base64 decoding differs between Evolution, SpamAssassin, and

ClamAV when multiple lines of base64 data do not start with a whitespace. Evolution parses line-by-line, SpamAssassin only decodes the first line of base64, and ClamAV decodes the entire body as one. Note that a single base64 blob can - for all three parsers that support base64 - be safely split over multiple lines if the extra lines are line-wrapped according to the IMF line wrapping method. Then, these three parsers will behave the same. This is odd, as the MIME standard is clear on how to handle unknown characters.

- **D9** (Fallback to Quoted-Printable): ClamAV uses Quoted-Printable decoding whenever the stated transfer encoding is invalid but the data contains =.
- **D13** (ClamAV "close-enough" base64): ClamAV can be convinced to do base64 decoding by considering a Content-Transfer-Encoding that looks a bit like `base64`, for example, `bas64` or `base6`. This limits ClamAV only to decode it as base64

Table II summarizes which parser(s) each of the root causes affects. We analyze exploitability in Section VII, focusing on those differentials most relevant to email smuggling scenarios. We also have Thunderbird in this table, which we will discuss later (in section VII).

Root cause	Postfix	SpamAssassin	ClamAV	Thunderbird
D1	✓	✗	✗	✗
D2	✓	✗	✗	✗
D3	Unknown[1]	✓	Unknown[2]	✗
D4	Unknown[3]	✓	✓	✗
D5	Unknown[4]	✓	✓	✗
D6	✗	✓	✗	✗
D7	✓	✗	✓	✓
D8	✓[1]	✓	✗	✗
D9	✗	✗	✓	✗
D10	✗	✗	✓	✗
D11	Unknown[2]	✓	Unknown[3]	✓
D12	✓	✓	✓	✓
D13	✗	✗	✓	✗
D14	✗	✓	Unknown[4]	✓
D15	✓	✓	✓	✓

[1] Due to **D2**.
[2] Due to **D1**.
[3] Unknown because ClamAV always does base64 decoding.
[4] Due to **D10**.

TABLE II: Overview of the root causes of parser differentials. The first three columns show the results of experiments in section VI-B, the last column of experiments in section VII-B.

C. Memory corruption vulnerabilities

In addition to uncovering the parser differentials described above, our differential fuzzing using AFL++ revealed numerous memory corruption vulnerabilities in the MIME parsers of Postfix, ClamAV, and Evolution. Upon further analysis, we found that many of these issues manifested as memory assertion errors; however, we also identified a significant number of memory corruption bugs. We are still in the process of analyzing the root causes of these vulnerabilities, and several have already been reported to the respective maintainers.

It is disappointing to see that there are some many memory corruption bugs in these parsers that could be found by simple fuzzing. One would hope that email handling code is fuzzed as part of the quality assurance process, especially since the code is handling data that comes from the public internet. Note that our fuzzing campaign was not even intended to look for these kinds of bugs; for instance, the code was not instrumented with sanitisers to help with finding memory corruption bugs.

Interestingly, Postfix, ClamAV, and Evolution are included Google's OSS-Fuzz initiative [8]. This means that they have already been fuzzed, including by AFL++, as this is one of the fuzzers used by OSS-Fuzz. Apparently the memory corruption vulnerabilities that we uncovered have not been found in the OSS-Fuzz fuzzing campaigns. Presumably the fuzzing harnesses (or entry points) provided by the projects for OSS-Fuzz are inadequate. So while it is a positive sign that the projects are enrolled with OSS-Fuzz, it is disappointing to see that this is not done with the care and attention it deserves.

VII. Exploitability analysis

This section discusses our tests to see if the discovered parser differentials are exploitable for email smuggling as defined in Section III, i.e. if they can be exploited to sneak an email or email attachment past a anti-virus or spam filter because the differential causes the filter to ignore it. We first describe the setup we used to analyze exploitability and then provide our results and discuss these.

A. Exploitability test set-up

To test for exploitability, we manually crafted emails that were designed to be blocked by ClamAV or flagged as spam by SpamAssassin but which might slip past these filters because of some parser differential. To craft these emails we used specific payloads that should either trigger ClamAV to block it as a virus or trigger SpamAssassin to mark it as spam. We used these payloads in combination with each of the (applicable) parser differentials we discovered. If the resulting email then ends up in the email client without being blocked or marked as spam then the differential is exploitable.

As test set-up we used a Postfix mail server with SpamAssassin and ClamAV as filters. We connected to this with two email clients, Evolution and Thunderbird, to see how these treat the emails. This results in a setup as shown in Figure 5.

The Postfix mail server tells us if an email is blocked by ClamAV or tagged as spam by SpamAssassin. If one of the email viewers, Evolution or Thunderbird, shows data that should have been tagged as spam by SpamAssassin without marking at such, or provides data that should have been blocked by ClamAV, we have an exploitable parser differential for that viewer.

The payloads we used are:

1) The EICAR anti-malware test value[3] - for ClamAV, and

[3]https://www.eicar.org/download-anti-malware-testfile

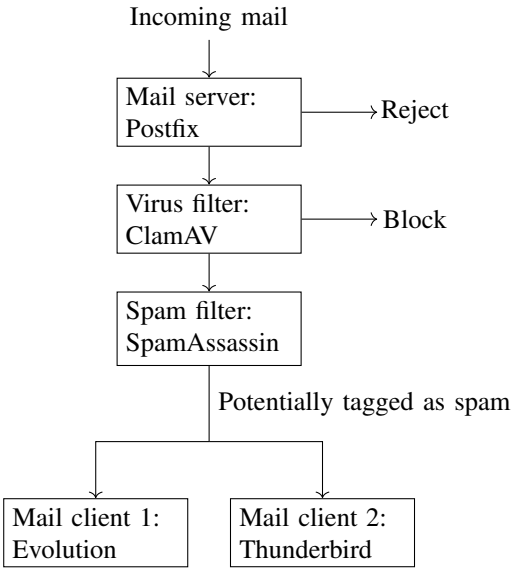

Fig. 5: Our exploitability test setup

2) The GTUBE spam filter test value[4] - for SpamAssassin.

These are values that should trigger ClamAV and SpamAssassin, respectively: ClamAV should block the EICAR value, and SpamAssassin should mark any email containing the GTUBE value as spam. Depending on the parser differential we try to exploit, these values are sometimes used in the body, either as plaintext or base64-encoded, or in the MIME multipart-preamble.

The EICAR test values have several versions, of which we use two:

1) *The plain-text value:* This is easier to use than the zip file version but is limited because it cannot be a substring of a larger text. This means that differentials that require more data at the start or end will stop ClamAV from detecting this value.
2) *A zip file test:* This is more difficult to use than the plain-text value because it has to be encoded to be used in a valid email. This results in more complicated tests but does allow for more data at the end of the file - because then the zip file is still valid. It still does not allow for more data at the start of the file, as the zip file is no longer valid.

Whenever a difference does not require additional data at the start or end of the test value, we use the plain-text version. We use the zip file version when the difference requires additional data at the end of the test value. When the difference require additional data at the start of the test value, we cannot use either test value and thus can not test for exploitability.

Postfix also contains some filters, namely to validate the email format. Obviously emails that are rejected by Postfix as invalid are not interesting as test cases for smuggling data past ClamAV and SpamAssassin.

[4]https://spamassassin.apache.org/gtube

B. Exploitability test results

Using the manually crafted emails in the set-up as discussed above, we found that some of the parser differentials in Section VI-B are exploitable. An overview of which differential is exploitable for smuggling data past SpamAssassin or ClamAV when using either Evolution and Thunderbird as mail client is given in Table III. They are discussed in more detail below. For some differentials, marked as Unknown in Table III, we could not make sure because of our testing setup if they really were exploitable or not, as we explain this in the reminder of this section.

The fact that some of exploits work for Evolution but not for Thunderbird shows that there are also parser differentials between these email viewers. We will publish the exploit emails once the vulnerabilities have been addressed by the respective vendors, ensuring a responsible disclosure process while still providing the community with reproducible test cases.

1) Difference D3: For SpamAssassin, difference D3 is exploitable by putting a base64 transfer encoding header first and a 7bit transfer encoding header second. Testing this with the GTUBE value shows that the email does not get marked as spam, even though both Evolution and Thunderbird show the GTUBE value. This is a successful email smuggling.

For ClamAV, this difference is not exploitable because ClamAV still blocks the email.

2) Difference D4: For SpamAssassin, difference D4 is exploitable. It can be exploited by using a plain text content type header first, then using a multipart alternative content type header second, and putting the GTUBE value in a multipart formatted message in the multipart-preamble. The GTUBE value appears on the first line in Thunderbird and Evolution.

We note an entry in the logs about the mail containing bad headers, though this does not impact the mail being delivered.

For ClamAV, this difference is not exploitable because ClamAV still blocks the email.

3) Difference D5: difference D5 (difference in base64 decoding) is not exploitable through SpamAssassin.

Testing this difference requires more data at the start of the test value, so we cannot use our setup to test if it is exploitable for ClamAV. This is a limitation of the EICAR test values as explained in VII-A.

4) Difference D7: SpamAssassin marks the email as spam, so it is not exploitable through SpamAssassin.

We cannot test whether ClamAV is exploitable, as it requires more data at the start of the test value. This is a limitation of the EICAR test values - as explained in VII-A. This means that we do not know if this is exploitable for ClamAV.

We also realized that this difference occurs between Thunderbird and Evolution. Thunderbird also parses the header prefixed with a carriage return as part of the body of the mail, like ClamAV and Postfix.

5) Difference D9: Difference D9 is not exploitable through SpamAssassin, but we cannot test this with our test values for ClamAV. That is because the test values do not contain equal signs and do not allow for modifications to include them.

Difference cause	Evolution	Thunderbird
D1	✗	✗
D2	✗	✗
D3	✓	✓
D4	✓	✓
D5	✗	✗
D6	✗	✗
D7	✗	✗
D8	✗	✗
D9	✗	✗
D10	✗	✗
D11	✓	✗
D12	✗	✗
D13	✗	✗
D14	✗	✗
D15	✗	✗

(a) Smuggling through SpamAssassin

Difference cause	Evolution	Thunderbird
D1	✗	✗
D2	✗	✗
D3	✗	✗
D4	✗	✗
D5	Unknown	Unknown
D6	✗	✗
D7	Unknown	Unknown
D8	✗	✗
D9	Unknown	Unknown
D10	✗	✗
D11	✗	✗
D12	✗	✗
D13	✗	✗
D14	✗	✗
D15	✗	✗

(b) Smuggling through ClamAV

TABLE III: Overview of exploitability of parser differentials

6) Difference D11: Difference D11 is exploitable through SpamAssassin, but only for the Evolution mail client. The email to exploit it has an invalid base64 Content-Transfer-Encoding header, and then the GTUBE value base64 encoded in the body. Evolution still decodes the body, even though the header is invalid. Thunderbird does not decode the body and thus only shows the base64 encoded value.

7) Difference D15: Because the email is extensively malformed (multiple extra colons, control characters, truncated boundary markers), Postfix lumps most invalid lines into the message body (similar to D2) rather than rejecting them outright. However, ClamAV and SpamAssassin still see the complete payload and detect EICAR or GTUBE as intended, so D15 is not exploitable in our setup. Meanwhile, Evolution and Thunderbird display differentials (with Evolution attempting to "repair" more headers), illustrating how heavily corrupted headers can lead to divergent client-side behavior—even if they do not bypass spam or antivirus filters.

C. Discussion on the exploitability test results

For ClamAV, we were limited by the EICAR values that cannot have more data at the start. Due to this, we could not test the exploitability of three of the nine parser differentials we found for ClamAV, so exploitability of these is unknown.

Only three of the eight parser differentials we found were exploitable for SpamAssassin. One of those was only exploitable with the Evolution mail client, not with Thunderbird. While this means we have succeeded in smuggling data through SpamAssassin, it is disappointing – from an attacker's perspective – that so many parser differentials we found are not exploitable for SpamAssassin and ClamAV.

We expect this to be either due to a different part of the full setup – which may change the contents of the mail subtly – or it could be that SpamAssassin and ClamAV have some output that is not picked up by the harness (they may, for example, also look at the data before it enters the MIME parser). In either case, it is good to see – from a defender's perspective – that this means there are few methods for successfully smuggling data.

The differentials we found between the two email clients Thunderbird and Evolution (D7, D11, D12, D14) are also interesting. They show that the mail clients do not always behave the same either, which can be used to show different data to different users - depending on which mail client they use. Since Thunderbird was not part of our differential fuzzing campaign, we did not expect to find such differentials. It shows that the differential fuzzing produced examples of complicated corner cases that not just problematic for the parsers used in the fuzzing, but also for other parsers.

VIII. Related Work

A. Email Smuggling

There is some prior research into MIME or SMTP differentials that might allow attackers to bypass security checks. But, as discussed below, none of that work involved the use of fuzzing to automatically find differentials. Also, all this earlier work looked at different aspects of the email protocol stack, namely authentication possibilities.

Chen et al. [9] attempted to circumvent email authentication (SMTP/DKIM) by exploiting differentials between mail servers and clients. Likewise, Longin [10] focused on abusing SMTP to bypass certain security checks. Although these works illustrate the potential for message-based attacks, neither uses an automated fuzzing approach, and both focus on SMTP rather than MIME.

Müller et al. [11] studied how to spoof S/MIME signatures by manipulating signature data within MIME. Similarly, Poddebniak et al. [12] introduced Efail, which leverages exfiltration channels in S/MIME and OpenPGP to break email encryption through maliciously crafted MIME structures. While these attacks partially overlap with our goal of discovering vulnerabilities in MIME processing, they do not rely on systematic fuzzing, nor do our targets include a dedicated signature-checker parser (see Section IV). Hence, we cannot directly compare results, but they underline the broader risk of parsing inconsistencies in secure email systems.

Both Chen et al. [9] and Müller et al. [11] use manual analysis to find vulnerabilities in MIME structures. In contrast, our work automates the discovery of MIME parser differentials, using both a grey-box fuzzer (AFL++) and a black-box grammar-based fuzzer (T-Reqs). This approach systematically uncovers hidden differentials that manual analysis might miss.

B. Differential Fuzzing

There has been prior work on differential fuzzing for case studies other than email.

Numerous studies have applied differential fuzzing to different domains:

- URLs – Dippygram [13],
- Ethereum Virtual Machines – EVMFuzz [14],
- JavaScript Engines – Jit-picking [15],
- HTTP servers – T-Reqs [16], Gudifu [17],
- QUIC – DPI-Fuzz [18].

Some of these (Dippygram, Jit-picking, Gudifu) use coverage guidance, like our AFL++ approach. T-Reqs [16] and Gudifu [17] also focus on data smuggling (in HTTP contexts), mirroring our approach of bypassing email filters.

NEZHA [19] pioneered differential testing for semantic bugs, and HDiff [20] built on it to reveal Host header confusion and request smuggling. Chen et al. [21] and Nguyen et al. [22] similarly exposed HTTP parsing flaws leading to cache poisoning and Host header mismatches. Although these works focus on HTTP, we adopt a comparable methodology for MIME parsers, revealing that similar parser differentials also arise in email.

Finally, SFADiff [23] employs fuzzing to build Symbolic Finite Automata (SFAs) from multiple implementations (e.g., TCP stacks, web browsers) and compares them to identify differentials. While we compare concrete outputs for MIME, SFADiff could conceptually be adapted to model MIME parsers, offering insights via state machines rather than direct output comparisons.

Finally, SFADiff [23] employs fuzzing to build Symbolic Finite Automata (SFAs) from multiple implementations (e.g., TCP stacks, web browsers) and compares them to identify differentials. Active-learning approaches such as protocol state fuzzing for TLS [24], as well as recent work on analyzing control-plane protocols in 5G basebands [25], also rely on state-based testing and might be seen as specialized forms of differential fuzzing. While our work compares concrete outputs from MIME parsers, SFADiff (and related SFA-based methods) could conceptually be adapted to model MIME parsers, providing state-machine insights rather than direct output comparisons.

IX. Future Work

Because we used differential fuzzing specifically on the MIME parser, we were limited to targets where we could easily isolate the MIME parser. This limited our research to open source implementations. Proprietary targets, such as Proofpoint Email Protection[5] or Outlook[6](which are services), could be interesting targets to investigate too on further research. Our exploitability analysis could be performed on proprietary implementations, though care should be taken not to get blocked or put on blocklists. For this reason, we have not tried this.

We limited ourselves to the MIME parser, but as explained in section II, emails consist of more formats and protocols. Looking for parser differentials in IMF or SMTP could also be interesting for further research. Also, other additional formats for email—like SMIME—could be interesting to target with fuzzing. Additionally, we plan to finish analyzing and reporting the memory corruption vulnerabilities uncovered during our fuzzing campaign.

MIME parsers can have multiple outputs, but not all systems use all outputs. Mail clients, in particular, will only show a specific output to the user, but not all of them. A typical email client will either show the plaintext or the HTML part, but not both. It could also be interesting, separate from this research, if different mail clients handle this differently.

As explained in section V, our fuzzing harness uses coverage guidance instrumentation for the concatenation of our three targets written in C. Splitting the coverage guidance instrumentation per target could be interesting to filter out more similar testcases. To illustrate how multiple targets can create apparent duplicates, consider an example with two test cases (T1 and T2) that produce a differential only between ClamAV and Evolution. Suppose both T1 and T2 follow the same execution paths in ClamAV and Evolution (leading to the same parser difference), but T2 takes a different path in Postfix. Because our current instrumentation tracks all targets together, the harness sees T2 as *unique* based on Postfix's new path—even though, from the perspective of ClamAV and Evolution (where the actual differential occurs), T1 and T2 are essentially the same case. Splitting instrumentation per target and ignoring coverage changes for targets that do not exhibit a difference would reduce these unnecessary duplicates.

It is also possible to improve our differential grey-box fuzzer by adding support for structure-aware fuzzing by considering the input formats of the MIME specification. For example, the "Protocol Buffers" input format [26] with its mutator tool [27] can be used to enable structure-aware fuzzing.

X. Conclusion

We applied differential grey-box fuzzing to uncover many parser differentials between the MIME parsers in ClamAV, Postfix, SpamAssassin, and Evolution, a few of which could be exploited for email smuggling. Our approach automatically provided 448 example testcases that trigger parser differentials between some of the targets. Analyzing these differentials, we found 15 root causes, three of which turned out to be exploitable in the sense that they could bypass virus or spam filters. Here we only considered one particular setup of an

[5]https://www.proofpoint.com/us/products/email-security-and-protection/email-protection

[6]https://outlook.com

email system; other combinations of email servers, security filters, and clients may exhibit other vulnerabilities.

Our test cases also revealed some differentials between the ways the two email clients Evolution and Thunderbird parse MIME, even though Thunderbird was not part of the targets in our differential fuzzing campaigns. This shows our fuzzer found corner cases in the MIME format that other parsers might struggle with too.

Apart from the parser differentials we also found many memory corruption vulnerabilities in the C-based targets, especially in the fuzzing campaign with AFL++, even though it was not even our intention to look for these. What makes this surprising is that the targets in question are enrolled in OSS-Fuzz, so they have been fuzzed before, including with AFL++, but apparently poorly. Clearly the fuzzing harnesses for these targets supplied to let them be fuzzed by OSS-Fuzz can and should be improved. As these systems are internet-facing they should be rigorously fuzzed.

Grey-box fuzzing has the well-known advantage of automatically discovering interesting test cases, but for our research another important benefit was its ability to remove duplicate findings by only retaining those that trigger new execution paths. This was crucial because we had to manually inspect differentials to understand their root causes and manually construct examples that attempted to exploit them. Even when using the grammar-based fuzzer T-Reqs we needed AFL++'s corpus minimisation tool to reduce the huge number of differential examples from many thousands to several hundred. Even analyzing a few hundred testcases was already very labor-intensive.

While a grey-box fuzzer can discover more differentials in C-based targets, we have observed that T-Reqs—a grammar-based black-box fuzzer—found more differentials tied to header confusion and thus proved especially useful for email smuggling. This systematic mutation of MIME headers and encodings, unrestrained by instrumentation, leads us to conclude that a grammar-based grey-box fuzzer (similar to [28]) might be an even more effective strategy.

Our findings underline the challenges posed by the ambiguity of the MIME specification, which clearly leaves room for inconsistent interpretations among parsers. We have filed bug reports for parser differentials that can lead to email smuggling, though not all of these differentials indicate a specific flaw in one parser; in some cases, the cause is an inherent ambiguity in the MIME standard itself. Ideally of course, a more formal, unambiguous specification for data formats such as MIME would prevent – and maybe even eliminate – such parsing differentials and remove or at least limit the potential for exploiting them.

Acknowledgements

This research was funded by the Dutch Research Council (NWO) through the INTERSECT project (grant no. NWA.1160.18.301). The authors gratefully acknowledge this support.

References

[1] D. J. C. Klensin, "Simple Mail Transfer Protocol," RFC 2821, Apr. 2001. [Online]. Available: https://www.rfc-editor.org/info/rfc2821

[2] OWASP. [n.d.]. HTTP Response Splitting. Retrieved on 07 Oct. 2024. [Online]. Available: https://owasp.org/www-community/attacks/HTTP_Response_Splitting

[3] D. Kaminsky, M. L. Patterson, and L. Sassaman, "PKI layer cake: New collision attacks against the global X.509 infrastructure," in *International Conference on Financial Cryptography and Data Security*, ser. LNCS, vol. 6054. Springer, 2010, pp. 289–303.

[4] CWE-444: Inconsistent Interpretation of HTTP Requests ('HTTP Request/Response Smuggling'). Retrieved on 21 Nov. 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/444.html

[5] Obfuscated Files or Information: HTML Smuggling. Retrieved on 21 Nov. 2023. [Online]. Available: https://attack.mitre.org/techniques/T1027/006/

[6] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[7] M. Meyers, "Fuzzing for differences between MIME parsers," 2024, master thesis, Radboud University.

[8] K. Serebryany, "OSS-Fuzz – Google's continuous fuzzing service for open source software," in *USENIX Security*, 2017.

[9] J. Chen, V. Paxson, and J. Jiang, "Composition Kills: A Case Study of Email Sender Authentication," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[10] T. Longin. SMTP Smuggling - Spoofing E-Mails Worldwide. Retrieved on 18 Dec. 2023. [Online]. Available: https://sec-consult.com/blog/detail/smtp-smuggling-spoofing-e-mails-worldwide/

[11] J. Müller, M. Brinkmann, D. Poddebniak, H. Böck, S. Schinzel, J. Somorovsky, and J. Schwenk, ""Johnny, you are fired!" – spoofing OpenPGP and S/MIME signatures in emails," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[12] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk, "Efail: Breaking S/MIME and OpenPGP email encryption using exfiltration channels," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 549–566.

[13] B. Kallus and S. W. Smith, "dippy_gram: Grammar-Aware, Coverage-Guided Differential Fuzzing," retrieved on 13 Aug. 2023.

[14] Y. Fu, M. Ren, F. Ma, X. Yang, H. Shi, S. Li, and X. Liao, "EVMFuzz: Differential fuzz testing of Ethereum virtual machine," *Journal of Software: Evolution and Process*, 2023.

[15] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-Picking: Differential Fuzzing of JavaScript Engines," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022.

[16] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-Reqs: HTTP Request Smuggling with Differential Fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1805–1820.

[17] B. Jabiyev, A. Gavazzi, K. Onarlioglu, and E. Kirda, "Gudifu: Guided differential fuzzing for HTTP request parsing discrepancies," in *RAID'24*. ACM, 2024.

[18] G. S. Reen and C. Rossow, "DPIFuzz: A differential fuzzing framework to detect DPI elusion strategies for QUIC," in *36th Annual Computer Security Applications Conference (ACSAC), 2020*. ACM, 2020.

[19] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on security and privacy (SP)*. IEEE, 2017, pp. 615–632.

[20] K. Shen, J. Lu, Y. Yang, J. Chen, M. Zhang, H. Duan, J. Zhang, and X. 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)*. IEEE, 2022, pp. 1–13.

[21] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of troubles: Multiple host ambiguities in HTTP implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1516–1527.

[22] H. V. Nguyen, L. L. Iacono, and H. Federrath, "Your cache has fallen: Cache-poisoned denial-of-service attack," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1915–1936.

[23] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias, "SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-Box Differential Automata Learning," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[24] J. De Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 193–206.

[25] K. Tu, A. Al Ishtiaq, S. M. M. Rashid, Y. Dong, W. Wang, T. Wu, and S. R. Hussain, "Logic gone astray: A security analysis framework for the control plane protocols of 5G basebands," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3063–3080.

[26] Protocol Buffers. [n.d.]. Protocol Buffers - Google's data interchange format. Github Repository. Retrieved on 07 Oct. 2024. [Online]. Available: https://github.com/protocolbuffers/protobuf

[27] Google. [n.d.]. libprotobuf-mutator. Github Repository. Retrieved on 07 Oct. 2024. [Online]. Available: https://github.com/google/libprotobuf-mutator

[28] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.