ParDiff: Practical Static Differential Analysis of Network Protocol Parsers

MINGWEI ZHENG, Purdue University, USA
QINGKAI SHI, Purdue University, USA
XUWEI LIU, Purdue University, USA
XIANGZHE XU, Purdue University, USA
LE YU, Purdue University, USA
CONGYU LIU, Purdue University, USA
GUANNAN WEI, Purdue University, USA
XIANGYU ZHANG, Purdue University, USA

Countless devices all over the world are connected by networks and communicated via network protocols. Just like common software, protocol implementations suffer from bugs, many of which only cause silent data corruption instead of crashes. Hence, existing automated bug-finding techniques focused on memory safety, such as fuzzing, can hardly detect them. In this work, we propose a static differential analysis called ParDiff to find protocol implementation bugs, especially silent ones hidden in message parsers. Our key observation is that a network protocol often has multiple implementations and any semantic discrepancy between them may indicate bugs. However, different implementations are often written in disparate styles, e.g., using different data structures or written with different control structures, making it challenging to directly compare two implementations of even the same protocol. To exploit this observation and effectively compare multiple protocol implementations, ParDiff (1) automatically extracts finite state machines from programs to represent protocol format specifications, and (2) then leverages bisimulation and SMT solvers to find fine-grained and semantic inconsistencies between them. We have extensively evaluated ParDiff using 14 network protocols. The results show that ParDiff outperforms both differential symbolic execution and differential fuzzing tools. To date, we have detected 41 bugs with 25 confirmed by developers.

CCS Concepts: • **Software and its engineering** → **Software defect analysis**; **Automated static analysis**; • **Security and privacy** → **Web protocol security**.

Additional Key Words and Phrases: Network protocol, protocol format specification, static program analysis, differential analysis.

ACM Reference Format:
Mingwei Zheng, Qingkai Shi, Xuwei Liu, Xiangzhe Xu, Le Yu, Congyu Liu, Guannan Wei, and Xiangyu Zhang. 2024. ParDiff: Practical Static Differential Analysis of Network Protocol Parsers. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 137 (April 2024), 27 pages. https://doi.org/10.1145/3649854

Authors' addresses: Mingwei Zheng, Purdue University, West Lafayette, USA, zheng618@purdue.edu; Qingkai Shi, Purdue University, West Lafayette, USA, shi553@purdue.edu; Xuwei Liu, Purdue University, West Lafayette, USA, liu2598@purdue.edu; Xiangzhe Xu, Purdue University, West Lafayette, USA, xu1415@purdue.edu; Le Yu, Purdue University, West Lafayette, USA, yu759@purdue.edu; Congyu Liu, Purdue University, West Lafayette, USA, liu3101@purdue.edu; Guannan Wei, Purdue University, West Lafayette, USA, guannanwei@purdue.edu; Xiangyu Zhang, Purdue University, West Lafayette, USA, xyzhang@cs.purdue.edu.

ACM 2475-1421/2024/4-ART137
https://doi.org/10.1145/3649854

1 INTRODUCTION

Network protocols are crucial for systems that require communication, such as robotic systems and the Internet of Things, to name a few. Network protocols specify the formats of communication messages and the steps that multiple parties must follow in order to communicate. Different implementations have been independently developed for the same network protocol, because of either historical reasons or specific design requirements such as reducing the energy consumption for an embedded system. For example, there are hundreds of different implementations for more than 15 protocols in the Bluetooth family.[1,2]

Message parser is a critical component of a protocol implementation, responsible for parsing network messages and checking their validity. Unfortunately, network message parsers are error-prone. Numerous bugs have been discovered, which often result in severe system failures, security breaches, and data loss. For example, Heartbleed [Heartbleed 2020], a buffer over-read bug discovered in the Transport Layer Security (TLS) protocol parser of OpenSSL in 2014, affects millions of web servers around the world.[3] This vulnerability has allowed attackers to extract sensitive information from servers due to a missing check of the message buffer bounds.

Among numerous bugs in message parsers, a number of them are silent data corruptions such that they violate protocol-specific properties but do not cause crashes. For example, the vulnerability, CVE-2022-26129,[4] which is detected by our approach and detailed in the next section, reads data beyond a protocol-specified range in an oversized buffer but does not access data beyond the buffer bounds. As such, it is not a common buffer over-read bug that can cause system crashes. Traditional static or dynamic bug-finding techniques, e.g., symbolic execution [Cadar et al. 2008; Shi et al. 2018; Wei et al. 2023; Xie and Aiken 2005], model checking [Ball et al. 2011; Cho et al. 2013; Musuvathi and Engler 2004], or fuzzing [Godefroid et al. 2012; Haller et al. 2013; Huang et al. 2020], cannot detect it unless they are provided with the protocol-specific oracles. Unfortunately, such specific oracles are often not available or entail substantial and error-prone manual efforts, because network protocols are usually specified in natural language documents and lack formal specifications.

To address the oracle problem, differential analysis [Arnaboldi 2023; Badihi et al. 2020; Johnson et al. 2011; Ma et al. 2018; Mora et al. 2018; Person et al. 2008; Verdoolaege et al. 2012] and testing [Bao et al. 2016; Zou et al. 2021] could be useful as they can effectively find domain-specific bugs by comparing different versions or implementations of a system. However, existing approaches have notable limitations. First, in terms of static analysis, approaches like graph differentiation [Johnson et al. 2011] and differential symbolic analysis [Person et al. 2008; Rutledge and Orso 2022; Verdoolaege et al. 2012] are limited to analyzing syntactically similar programs, e.g., programs evolved from the same base version. If programs are substantially different in their syntactic structures, these approaches may produce considerable false positives. Second, in terms of dynamic analysis, while existing differential testing (or fuzzing) [Arnaboldi 2023; Churchill et al. 2019] is capable of finding semantic differences in independent projects, their effectiveness strongly depends on the quality of their inputs and often suffers from low code coverage, leading to many false negatives. Additionally, using differential testing needs significant manual effort in bug diagnosing, since they are not designed to provide hints on bug locations. Therefore, developers must closely examine execution discrepancies and identify problematic code locations for each input.

Our Approach. This paper presents ParDiff, a novel static differential analysis that can discover silent and protocol-specific bugs in a top-down parser for network messages. ParDiff addresses

[1]https://wikipedia.org/wiki/List_of_Bluetooth_protocols
[2]https://wikipedia.org/wiki/Bluetooth_stack#Embedded_implementations
[3]https://en.wikipedia.org/wiki/Heartbleed
[4]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-26129

the limitations of prior work by abstracting protocol implementations to a high-level and unified representation. The unified representation leads to the precise comparison of fine-grained program behaviors, further enabling low-effort bug localization.

To identify semantic differences in two syntactically disparate implementations, ParDiff automatically transforms a protocol implementation into a finite-state machine (FSM) that characterizes the message formats. In this abstracted FSM, state transitions are conditioned by first-order logic formulas, which represent format constraints, and ordered by the indices of a message buffer. This high-level abstraction makes ParDiff effective, disregarding the syntactic disparity of multiple protocol implementations. Furthermore, the transformation discards all implementation details irrelevant to parsing and, thus, makes it possible for our tool to handle large programs.

To find bugs, ParDiff combines a bisimulation algorithm and SMT solvers to compare the FSMs extracted from different implementations. The bisimulation algorithm attempts to align state transitions in two FSMs so that we can compare the parsing status in a fine-grained way. That is, our differential analysis checks if the constraints on each pair of aligned state transitions are equivalent. Given that these implementations should follow the same protocol specification, the FSMs should precisely align with each other. Hence, any discrepancy in state transitions between these FSMs indicates a potential bug. This approach effectively breaks down the comparison of large, intricate formulas into multiple comparisons of smaller, manageable ones, enhancing the accuracy and precision of the analysis.

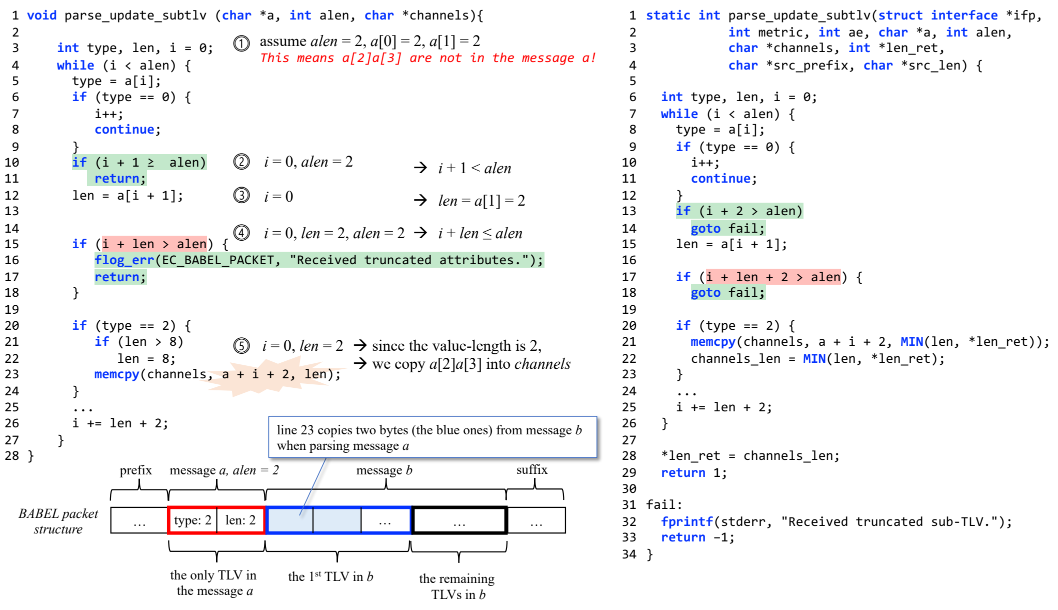
Finally, to reduce the manual effort in bug localization, ParDiff records source code locations during the process of format constraint abstraction. Consequently, every disparity detected in the FSMs can be directly traced back to its source code position, facilitating the precise identification of underlying issues. Notably, compared to common differential testing techniques, this process eliminates the need to scrutinize execution discrepancies across numerous inputs, significantly reducing the human effort to diagnose and localize bugs.

Contributions. In summary, we make the following contributions.

- We propose a novel static differential analysis for locating hidden bugs in protocol parsers.
  - It features a path constraint reduction algorithm to isolate the message parsing logic from other functionalities and to generate protocol format constraints.
  - It translates format constraints to a compact representation, i.e., constrained finite-state machines, for precise alignment between different implementations.
  - It leverages a bisimulation algorithm that precisely projects the differences in the constrained finite-state machines to buggy code at a fine-grained level.
- We implement the proposed approach in ParDiff,[5] a practical static differential analyzer for protocol parsers. We have evaluated ParDiff on 14 real-world protocols. The result shows that ParDiff is efficient and capable of analyzing two disparate implementations of a protocol in one minute on average. We have detected a total of 41 bugs, with 25 confirmed or fixed by the developers. Conventional differential symbolic execution cannot finish the analysis due to the path explosion problem. Differential testing can only detect 3 of them in the same time budget (i.e., less than 2 minutes) and can only detect 25 in 24 hours (i.e., using over 720× time cost).

Organization. Section 2 presents a real-world bug to motivate our solution. Section 3 provides an overview of our approach. Section 4 presents the detailed design of ParDiff. Section 5 presents the evaluation, where ParDiff is compared to both static and dynamic analysis tools. Section 6 discusses the related work. Section 7 concludes the paper.

[5]ParDiff is publicly available at https://github.com/zmw12306/ParDiff.



```
 1 void parse_update_subtlv (char *a, int alen, char *channels){
 2
 3     int type, len, i = 0;
 4     while (i < alen) {
 5       type = a[i];
 6       if (type == 0) {
 7          i++;
 8          continue;
 9       }
10       if (i + 1 ≥  alen)
11          return;
12       len = a[i + 1];
13
14
15       if (i + len > alen) {
16          flog_err(EC_BABEL_PACKET, "Received truncated attributes.");
17          return;
18       }
19
20       if (type == 2) {
21          if (len > 8)
22             len = 8;
23          memcpy(channels, a + i + 2, len);
24       }
25       ...
26       i += len + 2;
27     }
28 }
```

① assume $alen = 2, a[0] = 2, a[1] = 2$
This means a[2]a[3] are not in the message a!
② $i = 0, alen = 2$ → $i + 1 < alen$
③ $i = 0$ → $len = a[1] = 2$
④ $i = 0, len = 2, alen = 2$ → $i + len \le alen$
⑤ $i = 0, len = 2$ → since the value-length is 2, → we copy $a[2]a[3]$ into *channels*
line 23 copies two bytes (the blue ones) from message *b* when parsing message *a*
prefix
message *a, alen = 2*
message *b*
suffix
BABEL packet structure
...
type: 2
len: 2
...
...
...
the only TLV in the message *a*
the 1st TLV in *b*
the remaining TLVs in *b*

(a) Implementation I: frrouting/frr

```
 1 static int parse_update_subtlv(struct interface *ifp,
 2            int metric, int ae, char *a, int alen,
 3            char *channels, int *len_ret,
 4            char *src_prefix, char *src_len) {
 5
 6   int type, len, i = 0;
 7   while (i < alen) {
 8     type = a[i];
 9     if (type == 0) {
10       i++;
11       continue;
12     }
13     if (i + 2 > alen)
14       goto fail;
15     len = a[i + 1];
16
17     if (i + len + 2 > alen) {
18       goto fail;
19
20     if (type == 2) {
21       memcpy(channels, a + i + 2, MIN(len, *len_ret));
22       channels_len = MIN(len, *len_ret);
23     }
24     ...
25     i += len + 2;
26   }
27
28   *len_ret = channels_len;
29   return 1;
30
31 fail:
32   fprintf(stderr, "Received truncated sub-TLV.");
33   return -1;
34 }
```

(b) Implementation II: jech/babeld

Fig. 1. Motivating example. (a) A buggy implementation. The circled numbers and the message structure provide an example to show how the buggy code allows us to read bytes out of the scope of the message *a* when parsing the message. (b) A correct implementation.

2 MOTIVATING EXAMPLE

Before diving into the technical details, let us first use a real-world bug found by ParDiff to illustrate what bugs our new approach can discover and the limitations of existing work.

2.1 Buffer Accesses Offending Protocol-Specified Bound

We consider a bug found in an implementation of the BABEL network routing protocol (RFC 8966) [Chroboczek and Schinazi 2023]. The program mistakenly accesses a buffer at an index, which is out of the bound specified by the protocol, but still within the memory-safe bound due to an over-sized allocation. Therefore traditional crash-focused approaches face difficulty in detecting it.

Figure 1 shows two BABEL implementations that exhibit this bug. The first implementation (Figure 1(a)) is excerpted from the FRRouting Protocol Suite [Developers 2023], and the second (Figure 1(b)) is from the other reference implementation [Chroboczek 2023]. The two implementations define different APIs, and exhibit substantial syntactic and semantic differences, as highlighted in green for syntactic differences and red for semantic differences.

Line 15 of Figure 1a manifests the bug, which checks a less constrained condition diverging from the condition at line 17 in Figure 1b. As shown at the bottom of Figure 1a, a BABEL packet consists of a prefix, multiple messages, and a suffix. The function `parse_update_subtlv` in Figure 1 parses a message `a`, whose length is `alen`, into a list of type-length-value (TLV) items iteratively. Each loop iteration parses one TLV item. A TLV consists of a field `type` (line 5), a length field `len` (line 12), and the value field, e.g., `channels` (line 23), whose length is specified by the length field. The incorrect check (line 15) allows the parser to read bytes beyond the delimited length of the current message (line 23), i.e., extruding into the next message or the suffix of the current message.

This does not cause a buffer over-read error in the usual sense, because it does not exceed the allocated bound of the whole buffer. However, it is semantically not allowed to use the content of the second message to interpret the first as all messages should be independent.

To see a concrete failing case, let us consider a message where `a[0]=2`, `a[1]=2`, and `alen=2`, as shown at the bottom in Figure 1a. Since the length of the message is 2 and a TLV item in the message contains at least two bytes, i.e., the type and the length bytes, we can conclude that the message `a` contains only one TLV item whose length is 2. This further implies that the TLV item contains an empty value. However, the `len` field (i.e., `a[1]`) mistakenly specifies that there is a two-byte value. The incorrect check at line 15 fails to recognize this inconsistency, causing two bytes from the following message to be undesirably copied at line 23. In contrast, the second implementation in (b) correctly checks `i + len + 2 <= alen` at line 17, thus, can filter this invalid message.

2.2 Why Existing Works Fail

Traditional methods, including symbolic execution (e.g., [Cadar et al. 2008; Shi et al. 2018; Wei et al. 2023; Xie and Aiken 2005]), model checking (e.g., [Ball et al. 2011; Cho et al. 2013; Musuvathi and Engler 2004]), and fuzzing (e.g., [Godefroid et al. 2012; Haller et al. 2013; Huang et al. 2020]), cannot effectively detect this bug. Although the buggy program accesses bytes beyond the range of a message, it does not access bytes beyond the whole buffer. That is, it does not violate common memory-safety properties but a domain-specific correctness oracle, which is either unavailable or very expensive to obtain in practice.

To address the domain-specific oracle problem, dynamic and static differential analyses have been proposed in the literature. Differential testing or fuzzing [Arnaboldi 2023; Reen and Rossow 2020; Zou et al. 2021] feeds the same input into different implementations and compares their execution behaviors. For example, DPIFuzz [Reen and Rossow 2020] encodes the runtime program behavior into a hash and then compares the hashes generated by different implementations but with the same inputs. However, such techniques can hardly find this bug due to the following limitations. First, fuzzing techniques require high-quality seed inputs so that they can achieve high code coverage and generate (partially) valid protocol messages that can pass all preceding validation checks before reaching the buggy code. That is, in the aforementioned example (Figure 1), the message should be able to pass the checks at lines 6, 10, 15, and 20 in the first implementation. Since fuzzing techniques are mostly coverage-driven relying on random mutations, such a simple search strategy is unlikely to generate inputs satisfying the aforementioned conditions. In addition, they may generate a large amount of difference-inducing inputs. To comprehend the diverging execution behaviors, humans need to execute each input from the entry function of message parsers. This additional effort significantly increases the difficulty of identifying bugs from the differences.

Unlike fuzzing, as a static technique, differential symbolic analysis [Badihi et al. 2020; Mora et al. 2018; Person et al. 2008; Ramos and Engler 2011; Rutledge and Orso 2022; Verdoolaege et al. 2012] does not rely on seed inputs, and they can achieve high coverage of program paths and infer precise constraints over a packet. Due to the well-known path explosion issue in symbolic execution, these techniques often assume that two implementations have few differences and, thus, are primarily used to analyze multiple versions of the same software [Person et al. 2008]. In the example Figure 1 and more general situations, two protocol implementations can come from completely different codebases, containing a large number of differences in the code. Such settings break the assumption of differential symbolic analysis and make it impractical.

3 OUR APPROACH IN A NUTSHELL

In this section, we present an overview of ParDiff and illustrate its capability of detecting the bug in the motivating example. We also discuss how inherent challenges in our approach are addressed.

3.1 Overview

ParDiff automatically extracts message-parsing logic from different protocol implementations to finite state machines (FSMs) constrained by message-parsing properties. By lifting programs to FSMs, unessential implementation differences are abstracted away. Then, ParDiff constructs a bisimulation relation [Gentilini et al. 2003] between two FSMs corresponding to two different implementations. The discrepancy between the two FSMs can be reified to a concrete input, indicating a potential bug. Checking discrepancy or equivalence at the FSM level allows us to efficiently compare two implementations in quasi-linear steps and locate buggy code at a fine-grained level. Compared to existing work that attempts to find subtle bugs in protocol parsers, ParDiff has the following advantages:

- Unlike conventional static differential analysis that suffers from scalability, ParDiff abstracts away format-irrelevant code and, thus, is capable of handling large programs.
- Unlike conventional dynamic differential analysis that suffers from low code coverage or requires manual effort to recover protocol formats, ParDiff automatically infers precise protocol formats and can reach deep code guarded by complex conditions.
- ParDiff generates inputs that are precisely mapped to buggy source code locations, significantly reducing the human effort to diagnose and localize bugs.

3.2 How ParDiff Works

Despite these advantages, it is technically challenging to realize ParDiff in practice due to path explosion and fine-grained comparison of path conditions. Next, we briefly discuss the steps of ParDiff, together with our solution to addressing the inherent challenges.

Stage 1: Extracting Protocol Format Constraints (detailed in Section 4.1). A message parser imposes constraints on input messages. These constraints in an execution path encode a message format. Ideally, ParDiff should exhaustively enumerate all paths to generate complete protocol formats. However, this is challenging in practice, because there are an exponential number of paths for any nontrivial program, which is known as the path explosion problem. Moreover, protocol parsers may contain auxiliary code, i.e., routines that are not related to parsing or respectively only exist in one parser but not the others. Taking these auxiliary routines into account leads to false positives in comparison.

To mitigate path explosion and reduce false positives, ParDiff must only select a finite number of paths that are critical for identifying discrepancies or establishing the equivalence between two parsers. To this end, we first adopt loop unrolling and state merging, as commonly used in symbolic execution and bounded model checking. However, this is not enough since the extracted format constraints may still contain constraints irrelevant to the parsing logic. Thus, ParDiff also filters out constraints that are not imposed on the input buffer.

In the example of Figure 1, we unroll the loop once and only extract constraints imposed on the input buffer a. In Figure 1(a), one execution path X_1 goes through line numbers $4 \longrightarrow 6 \longrightarrow 8 \longrightarrow 4 \longrightarrow 28$, and is constrained by ϕ_{X_1}, i.e., $0 < alen \wedge a[0] = 0 \wedge 1 \geq alen$. The constraint forms a valid format for the set of messages that only contain a single byte of zero. Similarly, the constraint of the path $X_2 : 4 \longrightarrow 6 \longrightarrow 10 \longrightarrow 15 \longrightarrow 20 \longrightarrow 23 \longrightarrow 4 \longrightarrow 28$ is $\phi_{X_2} \equiv 0 < alen \wedge a[0] \neq 0 \wedge 1 < alen \wedge a[1] \leq alen \wedge a[0] = 2 \wedge a[1] + 2 \geq alen$, indicating a valid format where $a[0] = 2$. We can also extract similar constraints ϕ_{Y_1} and ϕ_{Y_2} from the second implementation (Figure 1(b)).

Stage 2: Comparing Protocol Format Constraints (detailed in Section 4.2). Assume ϕ_1 and ϕ_2 are two format constraints obtained from their corresponding implementations. To find differences between two protocol implementations, it seems straightforward to use an SMT solver to check if $\phi_1 \neq \phi_2$ is satisfiable. A satisfiable assignment indicates an input message that can be parsed

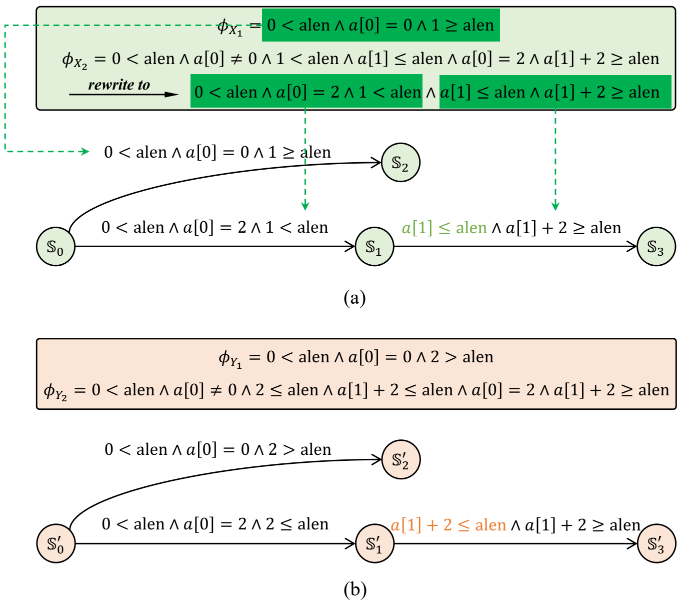

$\phi_{X_1} = 0 < \text{alen} \wedge a[0] = 0 \wedge 1 \geq \text{alen}$
$\phi_{X_2} = 0 < \text{alen} \wedge a[0] \neq 0 \wedge 1 < \text{alen} \wedge a[1] \leq \text{alen} \wedge a[0] = 2 \wedge a[1] + 2 \geq \text{alen}$
rewrite to
$0 < \text{alen} \wedge a[0] = 2 \wedge 1 < \text{alen} \wedge a[1] \leq \text{alen} \wedge a[1] + 2 \geq \text{alen}$
$0 < \text{alen} \wedge a[0] = 0 \wedge 1 \geq \text{alen}$
$\mathbb{S}_2$
$\mathbb{S}_0$
$0 < \text{alen} \wedge a[0] = 2 \wedge 1 < \text{alen}$
$\mathbb{S}_1$
$a[1] \leq \text{alen} \wedge a[1] + 2 \geq \text{alen}$
$\mathbb{S}_3$
(a)
$\phi_{Y_1} = 0 < \text{alen} \wedge a[0] = 0 \wedge 2 > \text{alen}$
$\phi_{Y_2} = 0 < \text{alen} \wedge a[0] \neq 0 \wedge 2 \leq \text{alen} \wedge a[1] + 2 \leq \text{alen} \wedge a[0] = 2 \wedge a[1] + 2 \geq \text{alen}$
$0 < \text{alen} \wedge a[0] = 0 \wedge 2 > \text{alen}$
$\mathbb{S}'_2$
$\mathbb{S}'_0$
$0 < \text{alen} \wedge a[0] = 2 \wedge 2 \leq \text{alen}$
$\mathbb{S}'_1$
$a[1] + 2 \leq \text{alen} \wedge a[1] + 2 \geq \text{alen}$
$\mathbb{S}'_3$
(b)

Fig. 2. Constraints and FSMs for the two implementations.

in one implementation but not the other, thereby finding a semantic difference. However, this monolithic satisfiable assignment does not directly inform us which specific lines of the program contribute to the semantic discrepancy. Additionally, to expose all possible discrepancies, we must keep querying the SMT solver for various assignments. Hence, this coarse-grained use of the SMT solver is ineffective in locating the discrepancy between the two implementations (see Section 5.4).

To precisely locate the buggy statements in the protocol parser, ParDiff must identify the specific constraints that lead to semantic differences. This entails finding a matching between two format constraints so that ParDiff can safely ignore those constraints that do not lead to semantic differences. To this end, ParDiff transforms each format constraint to an FSM, which encodes the spatial parsing logic of the protocol format. Edges are conditioned by constraints imposed on the input messages. Transitions between states are ordered by constrained buffer indices. That is, if one state transition is constrained by the i-th element of a network message, its immediately subsequent state transitions should be constrained by the $i + 1$-th element. This ensures that we can align two FSMs by the order of how they constrain the input memory buffer.

Figure 2 illustrates partial FSMs, which are produced based on the format constraints collected from the two implementations in Figure 1. Each state transition describes how one or multiple consecutive bytes are parsed and each path from the start state to the final state describes how a valid message is parsed.

Stage 3: Locating Implementation Differences and Bugs (detailed in Section 4.3). Given two FSMs produced in the previous stage, we leverage bisimulation [Gentilini et al. 2003] to compare two FSMs, which can efficiently find nonequivalent state transitions. For example, in Figure 2, we can establish a bisimulation between $\mathbb{S}_0 \longrightarrow \mathbb{S}_1$ and $\mathbb{S}'_0 \longrightarrow \mathbb{S}'_1$, since their transition conditions are equivalent; however, the transitions $\mathbb{S}_1 \longrightarrow \mathbb{S}_3$ and $\mathbb{S}'_1 \longrightarrow \mathbb{S}'_3$ do not bisimulate each other.

Note that the two FSMs in Figure 2 are already isomorphic *modulo transition conditions*. However, in practice, two FSMs can differ a lot in both structures and transition conditions, and they can still

$$
\begin{array}{rcll}
a, alen, x & \in & \textit{Variables} & \\
\textit{Parser } P & := & parse(a, alen, x_1, x_2, \ldots)\{\ S\ \} & \\
\textit{Value } v & := & x \mid c & \\
\textit{Expression } e & := & v \mid v_1 \oplus v_2 \mid a[v] & \\
\textit{Stmt } S & := & x \leftarrow e & \textbf{assignment} \\
 & \mid & abort() & \textbf{abort} \\
 & \mid & \textbf{if } (v)\ \{\ S_1\}\ \textbf{else}\ \{\ S_2\ \} & \textbf{branching} \\
 & \mid & S_1; S_2 & \textbf{sequencing}
\end{array}
$$

$$\oplus \in \{\ \wedge, \vee, +, -, \times, \div, >, \geq, <, \leq, =, \neq\ \}$$

Fig. 3. The syntax of the language protocol parsers.

be compared by bisimulation. Since each state transition in the FSMs contains only constraints over a few message bytes, when bisimulation fails, we can easily locate the implementation differences according to the differences in state-transition constraints. For example, the aforementioned state transitions differ in the constraints $a[1] \leq alen$ and $a[1] + 2 \leq alen$, which correspond to the buggy code at line 15 and line 17 in the first and the second implementations, respectively.

4 DESIGN

This section discusses the details of the three stages in our approach: (1) collecting protocol format constraints (Section 4.1), (2) translating format constraints into a finite state machine (Section 4.2), and (3) comparing state machines to locate possible bugs in different implementations (Section 4.3). At the end of this section, we discuss the soundness of our design (Section 4.4).

4.1 Collecting Format Constraints

To locate the implementation differences, ParDiff first collects format-relevant path constraints from multiple protocol implementations. To avoid path explosion, ParDiff unrolls loops (and recursive function calls) up to a constant number, and merges path constraints at the joint point of multiple paths, instead of enumerating each path in a program. To remove format-irrelevant constraints, ParDiff ignores the constraints not imposed on the input buffer.

4.1.1 Language of Protocol Parsers. To illustrate how ParDiff works, we use a small language that models a protocol parser. The syntax of the small language is defined in Figure 3. In the language, the entry of a protocol parser is a function *parse* that takes at least two parameters as the inputs. Following the motivating example (see Figure 1), the parameter `a` is an array of bytes representing the network message to parse. The parameter `alen` represents the length of the array. It may also take additional arguments to configure the parser. Values in the language could be either a variable or a constant. An expression could be a value, a binary expression, or an array access of the input message. A program statement in the parser could be an assignment to a variable, an abort statement (modeling errors e.g., `flog_err` in Figure 1), a conditional statement, or a sequential composition of two statements. In our language, *abort* terminates the program. In a parser, the message `a` is often read-only. Thus, we do not have statements to modify the input message.

The small language is loop-free; when implementing ParDiff, we follow the standard approach in bounded model checking [Biere et al. 2009] to unroll loops. While loop unrolling may introduce unsoundness, ParDiff is effective in terms of bug detection (instead of sound verification) as shown in our evaluation. The language also does not model pointers. In the implementation, we utilize an off-the-shelf pointer analysis [Sui et al. 2011]. The language does not model function calls for the sake of simplicity, since we can inline all functions into the main parser.

$$\begin{array}{llll}
\textit{Abstract Value } \hat{v} & := & \top & \textbf{format-irrelevant value} \\
 & \mid & c & \textbf{constant} \\
 & \mid & a[\hat{v}] & \textbf{byte in message} \\
 & \mid & alen & \textbf{message length} \\
 & \mid & ite(\hat{v}_1, \hat{v}_2, \hat{v}_3) & \textbf{if-then-else} \\
 & \mid & \hat{v}_1 \oplus \hat{v}_2 & \textbf{binary operation}
\end{array}$$

Fig. 4. Abstract values.

Before performing the analysis, ParDiff users need to annotate the protocol entry function, the input message buffer a and its length $alen$. In practice, identifying their location is relatively straightforward, since the parser entry usually closely follows certain network system calls (e.g., *recv*, *recvfrom*, and *recvmsg*), as shown in prior works [Caballero et al. 2009; Shi et al. 2023].

4.1.2 Format Constraints. Given a program in the language, we collect path constraints relevant to protocol formats, referred to as the format constraint, via static symbolic analysis. The analysis maps each variable to an abstract value $\hat{v}$ in Figure 4. Specifically, we use $\top$ to denote a format-irrelevant value, i.e., values that do not depend on the input buffer and its length. We use $a[\hat{v}]$ to represent a byte in a message and $alen$ the message length. The ite constructs represent an if-then-else constraint. With the abstract value defined, we then define the output of the analysis, i.e., the format constraint, below.

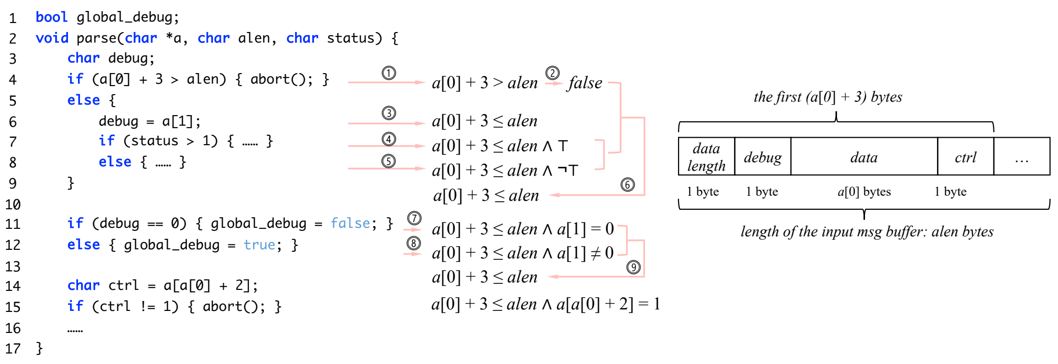
Definition 1 (Format Constraint) A format constraint ϕ is a formula over a set of atomic branching conditions[6] in a parser, satisfying the following requirements: (1) Each atomic branching condition in ϕ is a formula over $a[\hat{v}]$ and $alen$, without any format-irrelevant values. (2) Negating any atomic branching condition corresponds to a possible failure in the parsing procedure, i.e., triggering the abort statement.

- For requirement (1), a format constraint does not contain any format-irrelevant values (i.e., $\top$). During analysis, a format-irrelevant value is derived from a variable that is neither data-dependent nor control-dependent on the protocol message and, thus, is an implementation-specific variable, e.g., the variable `status` in Example 1 below.
- For requirement (2), each atomic condition in a format constraint must be related to some validity checks in the code. That is, negating the condition could lead to parsing failure, i.e., triggering the abort statement. Otherwise, the condition is not guarding against invalid format and, thus, is for other purposes that are irrelevant to parsing. For instance, for debugging, a parser implementation may include a branching statement *if* $(a[0] > 0)$ { *print*$(\ldots)$; }. Since there are no abort statements in both branches, such a branching condition $(a[0] > 0)$ does not imply the validity of a protocol message. Thus, ParDiff will exclude it in the format constraints.

Example 1 Figure 5 shows a parser, which has three inputs, the message `a`, its length `alen`, and a variable `status` that denotes the system status and is not related to parsing. The right part of the figure shows the structure of the network message. The first $a[0] + 3$ bytes can be split into four fields. The first field uses one byte and denotes the length of the third, the data, field. The second field uses one byte and determines if the system runs in debugging mode. The third is the `data` field, whose length is $a[0]$. The fourth is the `ctrl` field, whose value must be 1.

The parser contains four branching conditions at Lines 4, 7, 11, and 15. The format constraint is $a[0] + 3 \leq alen \wedge a[a[0] + 2] = 1$, which can be inferred from the branching conditions in line 4

[6]An atomic branching condition is a boolean expression used in an if-statement, and does not contain any boolean connectives (e.g., $\wedge$ or $\vee$). For example, $a[0] > 1$ and $a[1] > a[0]$ are two atomic conditions in **if** $(a[0] > 1 \ \wedge \ a[1] > a[0])$.



```
1  bool global_debug;
2  void parse(char *a, char alen, char status) {
3      char debug;
4      if (a[0] + 3 > alen) { abort(); }
5      else {
6          debug = a[1];
7          if (status > 1) { ...... }
8          else { ...... }
9      }
10
11     if (debug == 0) { global_debug = false; }
12     else { global_debug = true; }
13
14     char ctrl = a[a[0] + 2];
15     if (ctrl != 1) { abort(); }
16     ......
17 }
```

① $a[0]+3 > alen$ ② $false$
③ $a[0]+3 \le alen$
④ $a[0]+3 \le alen \wedge \top$
⑤ $a[0]+3 \le alen \wedge \neg\top$
⑥ $a[0]+3 \le alen$
⑦ $a[0]+3 \le alen \wedge a[1]=0$
⑧ $a[0]+3 \le alen \wedge a[1] \ne 0$
⑨ $a[0]+3 \le alen$
$a[0]+3 \le alen \wedge a[a[0]+2]=1$
the first (a[0] + 3) bytes
data length
debug
data
ctrl
...
1 byte
1 byte
a[0] bytes
1 byte
length of the input msg buffer: alen bytes

Fig. 5. The left part shows an example of collecting format constraints. The right part shows the structure of the network message to parse.

and line 15 as discussed in the next example. The condition at Line 7 is not relevant because it is not related to any byte in the input message. The condition at Line 11 is not relevant, either, because while the condition checks if $a[1] = 0$, neither branch aborts. That is, the validity of the message is not related to the value of $a[1]$. □

4.1.3 Collecting Format Constraints via Path Constraint Reduction. The static analysis for collecting the format constraints is described as the inference rules in Figure 6. The inference rules define how each statement in our language updates the program state in the form of $\lfloor \mathbb{A}, \phi \rfloor\ S\ \lfloor \mathbb{A}', \phi' \rfloor$. Here, $\lfloor \mathbb{A}, \phi \rfloor$ and $\lfloor \mathbb{A}', \phi' \rfloor$ are the program states before and after the execution of statement S, respectively. In a program state, ϕ is the collected format constraint, while the abstract store $\mathbb{A}$ maps a program variable x to its abstract value. Additionally, the lookup operation on $\mathbb{A}$ is also defined for constants: $\mathbb{A}(c) = c$ for any constant c, indicating constants retain their values. $\mathbb{A}(x \mapsto \hat{v})$ represents the abstract store updated with variable x now bound to a new abstract value $\hat{v}$.

The rule Init initializes the program state by mapping the variable *len* to the abstract value *alen* and mapping all variables not related to network messages to a format-irrelevant value. The assignment rules AssignVal, AssignBin, and AssignArr are quite standard. For instance, in the rule AssignBin, if the abstract values of the variables v_1 and v_2 are $\hat{v}_1$ and $\hat{v}_2$, respectively, the resulting abstract value will be $\hat{v}_1 \oplus \hat{v}_2$. These rules follow the exact semantics of these statements. The rule Abort resets the path constraint to *false* because it terminates the program and cannot reach the exit of the program. The rule Sequencing means that we analyze the program statements in order, using the postcondition of S_1 as the precondition of S_2.

The rule Branching assumes that the abstract value of the branching condition is $\hat{v}$, and given the path constraint ϕ before the if-statement, the analysis result of the true branch is $\lfloor \mathbb{A}_1, \phi_1 \rfloor$ and the result of the false branch is $\lfloor \mathbb{A}_2, \phi_2 \rfloor$. The program state after a branching structure could be in three cases. In the first two cases, one of the branches cannot reach the joint point after the branching. Thus, we directly use the analysis result of the other branch that can reach the joint point. A branch may not be able to reach the joint point due to two possible reasons: (1) the path constraint of that branch is unsatisfiable, or (2) the branch contains an *abort* statement that terminates the program. In the third case where both branches can reach the joint point, we merge two abstract stores where the new variables are guarded using the *ite* operator, meaning that if the condition $\hat{v}$ is true, the abstract value of u is $\mathbb{A}_1(u)$ or, otherwise, is $\mathbb{A}_2(u)$ s. The path constraint is the disjunction of the path constraints from the two branches.

$$\frac{\mathbb{A} = \emptyset \quad \phi = true}{\lfloor \mathbb{A}, \phi \rfloor \; parse(a, alen, x_1, x_2, \dots) \; \lfloor \mathbb{A}(x_i \mapsto \top), \phi \rfloor} \textsc{Init} \qquad \frac{\mathbb{A}(v) = \hat{v}}{\lfloor \mathbb{A}, \phi \rfloor \; x \leftarrow v \; \lfloor \mathbb{A}(x \mapsto \hat{v}), \phi \rfloor} \textsc{AssignVal}$$

$$\frac{\mathbb{A}(v_1) = \hat{v}_1 \quad \mathbb{A}(v_2) = \hat{v}_2}{\lfloor \mathbb{A}, \phi \rfloor \; x \leftarrow v_1 \oplus v_2 \; \lfloor \mathbb{A}(x \mapsto \hat{v}_1 \oplus \hat{v}_2), \phi \rfloor} \textsc{AssignBin} \qquad \frac{\mathbb{A}(v) = \hat{v}}{\lfloor \mathbb{A}, \phi \rfloor \; x \leftarrow a[v] \; \lfloor \mathbb{A}(x \mapsto a[\hat{v}]), \phi \rfloor} \textsc{AssignArr}$$

$$\frac{}{\lfloor \mathbb{A}, \phi \rfloor \; abort() \; \lfloor \mathbb{A}, \text{false} \rfloor} \textsc{Abort} \qquad \frac{\lfloor \mathbb{A}_1, \phi_1 \rfloor \; S_1 \; \lfloor \mathbb{A}_2, \phi_2 \rfloor \quad \lfloor \mathbb{A}_2, \phi_2 \rfloor \; S_2 \; \lfloor \mathbb{A}_3, \phi_3 \rfloor}{\lfloor \mathbb{A}_1, \phi_1 \rfloor \; S_1; S_2 \; \lfloor \mathbb{A}_3, \phi_3 \rfloor} \textsc{Sequencing}$$

$$\frac{\mathbb{A}(v) = \hat{v} \quad \lfloor \mathbb{A}, \phi \wedge \hat{v} \rfloor \; S_1 \; \lfloor \mathbb{A}_1, \phi_1 \rfloor \quad \lfloor \mathbb{A}, \phi \wedge \neg\hat{v} \rfloor \; S_2 \; \lfloor \mathbb{A}_2, \phi_2 \rfloor}{\lfloor \mathbb{A}, \phi \rfloor \; \textbf{if } (v) \; \{S_1;\} \; \textbf{else } \{S_2;\} \begin{cases} \lfloor \mathbb{A}_1, \phi_1 \rfloor & \phi_2 \equiv false \\ \lfloor \mathbb{A}_2, \phi_2 \rfloor & \phi_1 \equiv false \\ \lfloor \mathbb{A}_3, \phi_1 \vee \phi_2 \rfloor & otherwise \\ \qquad where & \mathbb{A}_3 = \mathbb{A}(u \mapsto ite(\hat{v}, \mathbb{A}_1(u), \mathbb{A}_2(u))) \\ & for\, u \in DOM(\mathbb{A}_1) \cap DOM(\mathbb{A}_2) \end{cases}} \textsc{Branching}$$

Fig. 6. Inference rules for collecting format constraints.

To compute a format constraint and remove irrelevant branching conditions, we apply the inference rules in Figure 6 together with a set of simplification rules, including but not limited to:

- Simplify $\hat{v} \vee \top$ ($\hat{v} \neq false$) into $\top$ and $\hat{v} \wedge \top$ ($\hat{v} \neq true$) into $\hat{v}$.
- Simplify any other formula containing $\top$, e.g., $\hat{v} + \top$ and $\hat{v} > \top$, into $\top$.
- Simplify $(\hat{v}_1 \wedge \hat{v}_2) \vee (\hat{v}_1 \wedge \hat{v}_3)$ into $\hat{v}_1 \wedge (\hat{v}_2 \vee \hat{v}_3)$ by distributivity.
- Simplify $\hat{v} \vee \neg\hat{v}$ into *true* by the law of excluded middle, and simplify $\hat{v} \vee \hat{v}$ or $\hat{v} \wedge \hat{v}$ into $\hat{v}$.
- ...

The first and second simplification rules define the operations on a format-irrelevant value $\top$. Intuitively, the first simplification rule preserves format-relevant constraints while the second one removes irrelevant ones. The two rules together ensure the first requirement in Definition 1 to be satisfied. The third and fourth simplification rules ensure the second requirement in Definition 1 to be satisfied. That is, given a branching condition $\hat{v}$, if neither the true branch nor the false branch aborts the program, this branching condition is not format-relevant. Thus, the fourth rule simplifies $\hat{v} \vee \neg\hat{v}$ into *true*. The third rule facilitates the use of the fourth rule. We illustrate the analysis procedure in the following example.

Example 2 (Continued) Figure 5 shows the ①-⑨ steps of computing the format constraint. Initially, the abstract value of `status` is set to a format-irrelevant value $\top$. In Line 4, the path constraint of the true branch is $a[0]+3 > alen$ (Step ①), which is then set to *false* due to the program-terminating abort statement (Step ②). The initial path constraint of the false branch is $a[0] + 3 \leq alen$ (Step ③).

Since `status` $= \top$, the branching condition in Line 7 is $\top$ as per the second simplification rule. The path constraints of the true and the false branches are $a[0]+3 \leq alen \wedge \top$ and $a[0]+3 \leq alen \wedge \neg\top$, respectively (Steps ④ and ⑤). Both of them can be simplified into $a[0] + 3 \leq alen$ as $\neg\top$ yields $\top$ (the second simplification rule) and $a[0]+3 \leq alen \wedge \top$ yields $a[0]+3 \leq alen$ (the first simplification rule). Thus, the resulting path constraint after Line 8 is $a[0] + 3 \leq alen$.

In Step ⑥, we compute the merged path constraint. Since the true branch at Line 4 cannot reach the joint point, the constraint at Line 10 inherits the one from the false branch, i.e., $a[0] + 3 \leq alen$. Similarly, the path constraints at Lines 11 and 12 are $a[0] + 3 \leq alen \wedge a[1] = 0$ and $a[0] + 3 \leq alen \wedge a[1] \neq 0$. After Line 12, we merge them into $a[0]+3 \leq alen$ according to the third and fourth simplification rule. As such, we get the format-relevant path constraint $a[0] + 3 \leq alen$ at Line 13. Similarly, after Line 15, we get one more constraint, i.e., $a[a[0] + 3] = 1$. Thus, the final format constraint is $a[0] + 3 \leq alen \wedge a[a[0] + 2] = 1$. □

LEMMA 4.1. *Given a program in the language defined in Figure 3, the static analysis produces a sound format constraint.*

PROOF. (Sketch.) The correctness of this lemma is implied by two facts. First, all inference rules in Figure 6 model the exact semantics of the program statements. Second, the simplification rules applied to the path constraint do not change its semantic meaning. □

4.2 From Format Constraints to FSM

As discussed in Section 2, it is challenging to locate implementation differences at a fine-grained level by directly comparing two constraints either syntactically or semantically. Thus, we transform a format constraint into an FSM. The FSM specifies how a network message is parsed byte by byte (or field by field) as well as the constraints each byte or field needs to satisfy (Note: a field is multiple consecutive bytes). As such, the problem of comparing two protocol implementations is reduced to the problem of comparing two FSMs, which can then be addressed by bisimulation [Sangiorgi 1998] and provides two advantages (see Section 4.3): (1) the implementation differences can be located at a fine-grained level; (2) the comparison can complete in quasi-linear steps [Gentilini et al. 2003].

In what follows, we first discuss how to transform an *ordered* format constraint to an FSM, then detail how we construct ordered format constraints from disordered format constraints.

4.2.1 From Ordered Format Constraint to FSM. We define the ordered format constraint below.

Definition 2 (Ordered Format Constraint) A format constraint is ordered if and only if, for any sub-formula $\phi_1 \land \phi_2 \land \cdots \land \phi_n$ in the format constraint, for all *top-level byte* $a[m] \in \phi_i$ and *top-level byte* $a[n] \in \phi_j$, we have $i < j \Leftrightarrow m < n$.

In the definition, a *top-level byte* $a[k]$ is used to form other format constraints, instead of used to compute byte indices. For example, given the format constraint $a[0]+3 \leq alen \land a[a[0]+2] = 1$, we say $a[0]$ in the first sub-constraint is a top-level byte but it is not at the top level in the second sub-constraint. Intuitively, if constraints are faithfully extracted from parser implementation, ordered constraints mean that the parser processes bytes in the message buffer in order (e.g., parsing byte 0 before parsing byte 1).

Example 3 The format constraint, $(a[1] = 1 \lor (a[0] > 1 \land a[1] = 3)) \land a[2] < 1$, is ordered. □

Example 4 The format constraint, $a[0] + 3 \leq alen \land a[a[0] + 2] = 1$, is ordered. □

Example 5 The format constraint, $(a[1] = 1 \lor (a[1] > 1 \land a[0] = 3)) \land a[2] < 1$, is not ordered. □

Definition 3 (FSM Format) An FSM format contains a set of states and a set of transitions between states. It represents the entire message format in a sequential order.

- *State:* Each state is represented by $\mathbb{S}_k$, where $k \geq 0$. The initial state $\mathbb{S}_0$ symbolizes the condition before any part of the message is parsed.
- *State transition:* A transition is represented as a tuple $(\mathbb{S}_i, \phi, \mathbb{S}_j)$, where $i \geq 0$, $j > i$, and ϕ is a format constraint. The transition from state $\mathbb{S}_i$ to state $\mathbb{S}_j$ occurs iff the constraint ϕ is satisfied.

Algorithm 1 shows how we recursively transform an *ordered* format constraint to the defined FSM representation. We will explain how to turn unordered constraints into ordered ones later in Section 4.2.2. As a convention, transitions in a path of an FSM should parse bytes in a network message in order. That is, given two consecutive transitions, $(\mathbb{S}, \phi, \mathbb{S}')$ and $(\mathbb{S}', \phi', \mathbb{S}'')$, ϕ and ϕ' should respectively constrain two exclusive ranges of bytes and the bytes in ϕ precede bytes in ϕ'. Given a disjunctive constraint, Algorithm 1 creates an FSM for each sub-formula in the constraint and returns the union of these FSMs (line 3). Given a conjunctive constraint, Algorithm 1 creates an FSM for each sub-formula and concatenates them by connecting each final state in an FSM to each

Algorithm 1: Build FSM from Ordered Format Constraint.

```
procedure fsm($\phi$)
    if $\phi \equiv \phi_1 \vee \phi_2 \vee \cdots$ then
        $\mathbb{M} \leftarrow$ fsm($\phi_1$) $\cup$ fsm($\phi_2$) $\cup \cdots$;
    else if $\phi \equiv \phi_1 \wedge \phi_2 \wedge \ldots$ then
        $\mathbb{M} \leftarrow$ fsm($\phi_1$) $\oplus$ fsm($\phi_2$) $\oplus \cdots$;
    else
        $\mathbb{M} \leftarrow \{(\mathbb{S}, \phi, \mathbb{S}')\}$;
    return $\mathbb{M}$;
```

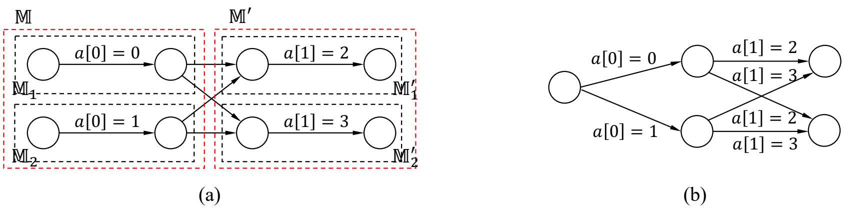


Fig. 7. (a) The finite state machine generated for the constraint $(a[0] = 0 \vee a[0] = 1) \wedge (a[1] = 2 \vee a[1] = 3)$. (b) The deterministic counterpart.

start state in the next FSM, with a transition constraint *true* (line 5). Given an atomic constraint ϕ that does not contain any connectives, i.e., $\wedge$ or $\vee$, we create a single state transition using ϕ as the transition constraint.

Example 6 This example illustrates how we translate the format constraint $(a[0] = 0 \vee a[0] = 1) \wedge (a[1] = 2 \vee a[1] = 3)$ into an FSM using Algorithm 1. As illustrated in Figure 7(a), the FSMs $\mathbb{M}_1$ and $\mathbb{M}_2$ are respectively generated for $a[0] = 0$ and $a[0] = 1$. The FSM $\mathbb{M}$ is the union of the two FSMs. The FSM $\mathbb{M}'$ is created in a similar way and is connected to the FSM $\mathbb{M}$. □

Lemma 4.2. *The FSM produced by Algorithm 1 is an equivalent representation of the input format constraint ϕ. That is, assuming the conjunction of the state-transition constraints on each path of the FSM (i.e., transitions from the start state to the final) is ϕ_i, we have $\phi \equiv \bigvee_i \phi_i$ for all ϕ_i.*

Proof. Given any format constraint ϕ, we use ϕ_i to represent the conjunction of all constraints in an FSM path (i.e., the state transitions from the start state to the final state).

Base: If a format constraint ϕ is an atomic constraint without any connectives $\wedge$ or $\vee$, Algorithm 1 returns from Line 7 where it generates a single state transition constrained by ϕ. In this case, it is apparent that the lemma is correct.

Induction: Consider two format constraints, γ and σ as well as their corresponding FSM, denoted as fsm(γ) and fsm(σ), which contain m and n paths, respectively. Let us assume that the lemma to prove is correct. That is, we have $\gamma \equiv \vee_{i=1}^{m} \gamma_i$ and $\sigma \equiv \vee_{i=1}^{n} \sigma_i$.

Induction Case (1): Consider the format constraint $\gamma \vee \sigma$, denoted as ϕ. As shown by line 3 in Algorithm 1, we have fsm(ϕ) = fsm(γ) $\cup$ fsm(σ), which, by definition, consists of two independent FSMs fsm(γ) and fsm(σ) and, thus, contains and only contains $m + n$ paths from fsm(γ)and fsm(σ). Thus, we have

$$\phi \equiv \gamma \vee \sigma \equiv \bigvee_{i=1}^{m} \gamma_i \vee \bigvee_{i=1}^{n} \sigma_i \equiv \bigvee_{i=1}^{m+n} \phi_i, \textbf{ where } \phi_i = \begin{cases} \gamma_i, & i \le m \\ \sigma_{i-m}, & i > m \end{cases}$$

Thus, if the lemma is correct for γ and σ, it is also correct for $\gamma \vee \sigma$.

Induction Case (2): Consider the format constraint $\gamma \wedge \sigma$, denoted as ϕ. As shown by line 5 in Algorithm 1, we have $\mathsf{fsm}(\phi) = \mathsf{fsm}(\gamma) \oplus \mathsf{fsm}(\sigma)$. In $\mathsf{fsm}(\phi)$, all final states of $\mathsf{fsm}(\gamma)$ are connected to all start states of $\mathsf{fsm}(\sigma)$. Hence, $\mathsf{fsm}(\phi)$ contains $m \times n$ paths, and $\phi_i = \gamma_j \wedge \sigma_k$, where $1 \le j \le m$ and $1 \le k \le n$. Therefore, we have

$$\phi \equiv \gamma \wedge \sigma \equiv \bigvee_{i=1}^{m} \gamma_i \wedge \bigvee_{i=1}^{n} \sigma_i \equiv \bigvee_{i=1}^{m\times n} \phi_i.$$

Thus, if the lemma is correct for γ and σ, it is also correct for $\gamma \vee \sigma$.

Summary: if the lemma to prove is correct for γ and σ, it is also correct for $\gamma \wedge \sigma$ and $\gamma \vee \sigma$. Thus, the lemma to prove is correct. □

As shown above, the generated FSM may be non-deterministic because there are the same transition constraints from one state to different states (e.g. the transitions connecting $\mathbb{M}$ and $\mathbb{M}'$ in Figure 7(a)). As a post-processing step, we follow existing automata theories [Khoussainov and Nerode 2012] to simplify each FSM and make it deterministic (as illustrated in Figure 7(b)). Note that as constraints are ordered to begin with, the state transition paths in the generated FSM are also ordered. This essentially allows us to align transition paths across FSMs from multiple implementations (e.g., aligning the transitions related to parsing the same byte in the message buffer) and conduct effective bi-simulation (see Section 4.3).

4.2.2 Reordering Format Constraints. In practice, a protocol parser may not rigorously follow the stream order to parse a message. As such, the format constraint collected in the first stage may not be ordered. In such cases, we employ Algorithm 2 to rewrite an arbitrary format constraint into an equivalent but ordered one.

In the algorithm, Lines 5-11 perform the main operation to reorder sub-formulas in a conjunctive constraint. Lines 5-7 are straightforward and transform a constraint like $a[1] > 1 \wedge a[0] > 1$ by switching the positions of $a[0] > 1$ and $a[1] > 1$. Lines 8-10 deal with a special case where two sub-formula cannot be reordered by switching positions. For instance, switching positions of $a[0] > a[2]$ and $a[1] > 1$ in the constraint $a[0] > a[2] \wedge a[1] > 1$ cannot make it ordered. In this case, we regard the two as a single atomic constraint by replacing $\wedge$ with an equivalent operator $\&$. Intuitively, we group constraints that cannot be ordered into *groups* such that groups can be ordered. For instance, the algorithm regards the three atomic constraints in $a[3] > 0 \wedge a[0] > a[2] \wedge a[1] > 1$ as two groups, one is $a[3] > 0$ and the other is $a[0] > a[2]$ & $a[1] > 1$. The two groups can be ordered by switching their positions, yielding $(a[0] > a[2]\ \&\ a[1] > 1) \wedge (a[3] > 0)$. When translating the constraint into an FSM, Algorithm 1 produces two consecutive transitions, one is constrained by $a[0] > a[2]\ \&\ a[1] > 1$ and the other is constrained by $a[3] > 0$. As such, the generated FSM satisfies the property that it parses a message in the stream order — in this example, it parses the first three bytes using the first state transition and then the fourth byte using the second transition.

Lemma 4.3. *The input and output format constraints of Algorithm 2 are equivalent.*

Proof. (Sketch.) Observe that in the algorithm, we only exchange the positions of two sub-constraints, e.g., ϕ_i and ϕ_j, if they are in a conjunctive formula. Apparently, $\phi_i \wedge \phi_j$ is equivalent to $\phi_j \wedge \phi_i$. Thus, the input and output format constraints of Algorithm 2 are equivalent. □

Lemma 4.4. *The output format constraint of Algorithm 2 is ordered.*

Algorithm 2: Reordering Format Constraint.

```
procedure reorder($\phi$)
    if $\phi \equiv \phi_1 \vee \phi_2 \vee \ldots$ then
        reorder ($\phi_1$); reorder ($\phi_2$); ...;
    else if $\phi \equiv \phi_1 \wedge \phi_2 \wedge \ldots$ then
        for top-level byte $a[m] \in \phi_i$, top-level byte $a[n] \in \phi_j$ such that $m$ and $n$ are the minimum indices in $\phi_i$
          and $\phi_j$, and $i < j \wedge m > n$ do
            switch the position of $\phi_i$ and $\phi_j$ in $\phi$;
        assume $\phi \equiv \phi_1 \wedge \phi_2 \wedge \ldots$ after reordering;
        for consecutive constraints, $\phi_j, \phi_{j+1}, \ldots$, such that byte indices in the constraints overlap do
            replacing $\wedge$ with an equivalent operator & in $\phi_j \wedge \phi_{j+1} \wedge \ldots$, meaning that they are grouped;
        assume $\phi \equiv \phi_1 \wedge \phi_2 \wedge \ldots$ after grouping;
        reorder ($\phi_1$); reorder ($\phi_2$); ...;
    else
        /* do nothing for constraints without $\wedge$ and $\vee$ */
```

Algorithm 3: Differentiating Formats by Bisimulation.

```
Procedure bisim(A pair of states $\mathbb{S}_1, \mathbb{S}_2$ from $\mathbb{M}_1, \mathbb{M}_2$)
    if one of $\mathbb{S}_1, \mathbb{S}_2$ is a final state then
        assume $\mathbb{S}_1$ is a final state and $\mathbb{S}_2$ is not;
        any outgoing constraint of $\mathbb{S}_2$ implies a difference between two implementations;
        return;
    foreach state transition $(\mathbb{S}_1, \phi_1, \mathbb{S}_1') \in \mathbb{M}_1$ do
        find $(\mathbb{S}_2, \phi_2, \mathbb{S}_2') \in \mathbb{M}_2$ such that $\phi_1 \equiv \phi_2$;
        if found then
            bisim($\mathbb{S}_1', \mathbb{S}_2'$);
        else
            find differences in $\phi_1$ and $\phi_2$;
```

Proof. (Sketch.) The correctness of this lemma is implied by two facts. First, Lines 5–10 in the algorithm reorder conjunctive formula strictly following Definition 2. Second, the recursive procedure in the algorithm reorders all conjunctive formulas in the input format constraint. □

4.3 Locating Implementation Differences

Algorithm 3 shows how we compare two FSMs by bisimulation. The basic idea of bisimulation is that starting from the start states of two FSMs, we try to match two states in two FSMs that can simulate each other. The matching is governed by the format constraint of transitions. As shown by lines 6-11 in Algorithm 3, given a state transition $(\mathbb{S}_1, \phi_1, \mathbb{S}_1')$ in one FSM, we try to find a transition $(\mathbb{S}_2, \phi_2, \mathbb{S}_2')$ in the other FSM such that $\phi_1 \equiv \phi_2$. If we find such an equivalent transition, we continue the bisimulation from the states $\mathbb{S}_1'$ and $\mathbb{S}_2'$ (line 9). If we cannot find such an equivalent transition, it means there exist some differences between the two protocol parsers under comparison. In this case, we will compare the outgoing transition constraints of the states $\mathbb{S}_1$ and $\mathbb{S}_2$ (line 11). During the bisimulation, if one of the FSMs reaches the final state but the other does not, it means that the two FSMs are not equivalent and implies implementation differences in the protocol parsers (see lines 2-5 in Algorithm 3). The correctness of Algorithm 3 is stated below, which is a direct result of the existing theory of bisimulation.

LEMMA 4.5. *Algorithm 3 guarantees to find the differences between a pair of state transitions in two FSMs if the two FSMs are not equivalent [Gentilini et al. 2003].*

Example 7 Consider the two FSMs in our motivating example (see Figure 2). We input the two start states, $\mathbb{S}_0$ and $\mathbb{S}'_0$ into Algorithm 3. For the two outgoing transitions from the state $\mathbb{S}_0$, we can respectively find two transitions in the other FSM from the state $\mathbb{S}'_0$ such that the transition constraints are equivalent (lines 6-7 in Algorithm 3). We then bi-simulate the two FSMs from $\mathbb{S}_1$ and $\mathbb{S}'_1$ (lines 8-9 in Algorithm 3). Since the two outgoing transitions from $\mathbb{S}_1$ and $\mathbb{S}'_1$ are not equivalent, we find the differences in the transition constraints, i.e., $a[1] \leq alen$ vs. $a[1] + 2 \leq alen$, to locate the implementation differences. □

From FSM Differences to Implementation Differences and Bugs. Since the transition constraints obtained from the last step correspond to branching conditions in the code, we can then locate the differences between the two implementations. For easier localization, we keep a record of the source code positions at which these constraints are generated during the first stage. As demonstrated in our motivating example (Section 2), such differences often imply some bugs in protocol parsers. Our evaluation shows that PARDIFF found 41 bugs, with 25 confirmed or fixed (Section 5.6).

4.4 Soundness

Lemmas 4.1–4.5 state that given protocol parsers written in the small language defined in Figure 3, our approach is sound and guaranteed to find differences between the parsers. The lemmas prove the following facts:

- Lemma 4.1 states that we generate a sound format constraint from the protocol parser;
- Lemmas 4.2–4.4 state that we translate each format constraint to an FSM, which is an equivalent representation of the format constraint;
- Lemma 4.5 states that by bisimulation, we can find the different state transitions if two FSMs are not equivalent.

In practice, we have to handle common program structures not included in the abstract language, which leads to a soundy [Livshits et al. 2015] implementation of PARDIFF. In other words, PARDIFF shares the same reasonable assumptions and standard approaches (to handle challenging program structures) with previous bug-finding techniques, e.g., [Babic and Hu 2008; Shi et al. 2018; Xie and Aiken 2005]. For example, in our implementation, we unroll each loop twice in the control flow graphs and call graphs. Following the aforementioned bug-finding techniques, we currently have not modeled inline assembly and call statements that invoke non-standard library APIs. For pointer analysis, we adopt Sui et al. [2011]'s approach to resolve pointer relations. The soundy (i.e., reasonably unsound) operations in the implementation do lead to false positives or negatives, which, however, are acceptable as we show in the evaluation. In total, PARDIFF lets us find 41 bugs in mature protocol implementations.

5 EVALUATION

We implement our tool PARDIFF on top of the LLVM (12.0.1) compiler infrastructure [Lattner and Adve 2004] and the Z3 (4.8.12) SMT solver [De Moura and Bjørner 2008]. The current implementation of PARDIFF works on protocol parsers written in C, but note that our approach is general for other common languages. Given a parser, we compile its source code into LLVM bitcode and send the bitcode to PARDIFF for further processing. In PARDIFF, Z3 is used to represent abstract values as symbolic expressions and check the equivalence of constraints. With the implementation, we design a series of experiments to answer the following four research questions:

- **RQ1**: How effective are the three stages of PARDIFF?

Table 1. Protocols and Their Codebases for Evaluation.

Protocols	**Codebases**	**Size**(loc)	**Description**
BABEL	FRR vs. BABEL	9K vs. 10K	Distance-vector routing protocol
BFD	FRR vs. BIRD	13K vs. 2K	Bidirectional forwarding detection
BGP	FRR vs. BIRD	2113k vs. 9k	Border gateway protocol
OSPF2	FRR vs. BIRD	72K vs. 14K	Open shortest path first
OSPF3	FRR vs. BIRD	38K vs. 14K	Open shortest path first v3
RADV	radvd vs. BIRD	14K vs.2K	Router advertisements
RIP1	FRR vs. BIRD	11K vs. 3K	Routing information protocol v1
RIP2	FRR vs. BIRD	11K vs. 3K	Routing information protocol v2
RIPng	FRR vs. BIRD	8K vs. 3K	Routing information protocol for ipv6
VRRP	FRR vs. Vrrpd	7K vs. 4k	Virtual router redundancy protocol
IPv4	lwip vs. Linux/ipv4	8K vs. 107K	Internet protocol v4
IPv6	lwip vs. Linux/ipv6	7K vs. 76K	Internet protocol v6
ICMP	lwip vs. Linux/ipv4	8K vs. 107K	Internet control message protocol
ICMP6	lwip vs. Linux/ipv6	7K vs. 76K	Internet control message protocol v6

- **RQ2**: How efficient are the three stages of ParDiff?
- **RQ3**: How effective is ParDiff in detecting bugs?
- **RQ4**: What are the root causes of the discovered bugs?

5.1 Experimental Setup

To create a set of protocols with multiple implementations, we refer to an index of open-source routing platforms [contributors 2022]. From this list, we apply a set of criteria to refine our selection: the projects must be implemented in C programming language and actively maintained within the past year. This results in the selection of FRRouting [community 2023] and BIRD [Martin et al. 2023], two open-source routing protocol suites. Next, for every protocol implemented in a suite, we find an alternate implementation that also meets the aforementioned criteria for comparison. Then we obtain a dataset consisting of ten network protocols, listed in the first ten rows in Table 1. Moreover, we incorporate four well-known protocols from the TCP/IP protocol suite. Finally, we get a dataset with 14 network protocols, each with two distinct implementations. The protocol and codebase information is shown in Table 1.

To answer RQ1, we run ParDiff on our dataset, and test the effectiveness of each stage. To evaluate the effectiveness of path constraint reduction (Stage 1), we record the number of LLVM instructions involved in computing the format constraint, with and without the constraint reduction algorithm. To evaluate the effectiveness of FSM generation and simplification (Stage 2), we record the number of FSM nodes and edges before and after FSM simplification. We then check the differences generated by the bisimulation to identify bugs (Stage 3). For each protocol we test, we record the number of differences, the number of differences caused by bugs, the number of differences caused by implementation options, and the number of unique bugs implied by the differences, in order to gauge the effectiveness of ParDiff. Additionally, we record the number of atomic branch conditions (cf. Definition 1) within each difference as a measure of human efforts in bug localization.

To answer RQ2, for each protocol, we record the execution time of each stage, together with the total time of our analysis, to measure the efficiency of our tool.

To answer RQ3, we evaluate our approach by comparing it to both static and dynamic analysis tools. In terms of static analysis tool comparisons, we conduct tests with existing static differential analysis, such as differential symbolic execution [Lahiri et al. 2012, 2013; Person et al. 2008, 2011; Ramos and Engler 2015; Rutledge and Orso 2022]. These techniques are indeed effective for identifying discrepancies across different versions of the same software by leveraging shared structures and code snippets. However, as discussed previously, they struggle with comparing independent protocol

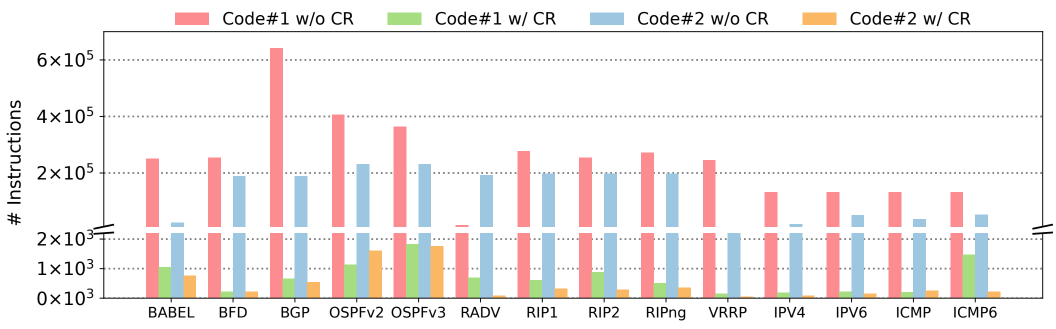


Fig. 8. Instructions used to compute format constraints, with and without constraint reduction (CR).

implementations, because the shared code structures that enable the efficiency of differential symbolic execution are mostly absent. As such, these tools need exhaustive and independent analysis on each implementation and fail to finish the analysis due to the path explosion problem. Thus, we omit the evaluation of these tools.

We also compare our fine-grained approach with the coarse-grained use of SMT solvers. The coarse-grained way monolithically queries the satisfiability of two format constraints ϕ_1 and ϕ_2, i.e., check if $\phi_1 \neq \phi_2$. If the query is satisfiable, we count the number of atomic branch conditions (cf. Definition 1) in both implementations, which is a measure of the manual effort to precisely locate the root cause (sub)condition in the program leading to the divergence.

As for dynamic analysis tools, we compare our tool with a differential fuzzing tool specially designed for protocol parsers, i.e., DPIFuzz [Reen and Rossow 2020]. While there are a few other differential fuzzing tools (e.g., [Petsios et al. 2017; Yang et al. 2021; Zou et al. 2021]), they are mostly domain-specific and have special designs for input generation and mutations, which cannot be directly applied to general network protocol parsers. Therefore, we select DPIFuzz, whose mutators are specifically designed for network protocol packets. We use the packet-level mutations and the execution behaviors (including abort or return in advance, return values of protocol parsers, etc) as fuzzing feedback. We run DPIFuzz with the first ten protocols in our dataset (Table 1), excluding IPv4, IPv6, ICMP, and ICMP6, which are kernel-space implementations (note that DPIFuzz is a user-space fuzzer that does not support fuzzing kernel code). For each protocol, we execute DPIFuzz in two settings: first, run with equal duration with ParDiff (that is, if ParDiff operates for x seconds, we also run DPIFuzz for x seconds). We repeat this procedure ten times to avoid random factors. Second, run each protocol for 24 hours, and repeat three times.

To answer RQ4, we analyze the root causes of all bugs we found and group them into three categories. We also provide case studies and discuss their potential impacts.

5.2 RQ1: Effectiveness of Each Stage in ParDiff

Stage 1: Collecting Format Constraints via Path Constraint Reduction. The numbers of LLVM instructions involved in computing format constraints, with and without our reduction algorithm, are shown in Figure 8. For all protocol implementations, the number of instructions has significantly decreased after applying path constraint reduction. This indicates that the path constraint reduction process is effective in reducing the complexity of the codebases, with an average of 99.57%.

Stage 2: From Format Constraints to FSM with Simplification. Table 2 indicates that the FSM simplification process in Stage 2 further reduces the complexity of each FSM after path constraint reduction in Stage 1. On average, there is an approximately 30.0% reduction in the number of

Table 2. Statistic of FSM node and edge number. Notably, the large differences in RADV and IPv6 protocols are due to incomplete implementations in the corresponding codebase.

Protocols	FSM 1		FSM 2	
	#Before Simplify	#After Simplify	#Before Simplify	#After Simplify
BABEL	1307 / 1318	641 / 652	538 / 557	227 / 240
BFD	19 / 20	19 / 20	71 / 72	65 / 66
BGP	47 / 50	46 / 49	121 / 150	74 / 86
OSPFv2	210 / 215	132 / 135	920 / 923	368 / 369
OSPFv3	367 / 428	127 / 188	544 / 543	374 / 373
RADV	421 / 466	370 / 395	4 / 4	4 / 4
RIP1	76 / 80	59 / 63	50 / 49	36 / 35
RIP2	75 / 76	44 / 44	19 / 21	19 / 21
RIPng	176 / 191	111 / 126	29 / 35	20 / 25
VRRP	48 / 48	48 / 48	15 / 14	15 / 14
IPV4	8 / 11	7 / 8	15 / 15	14 / 14
IPV6	6 / 5	4 / 3	358 / 376	150 / 158
ICMP	12 / 11	12 / 11	25 / 24	25 / 24
ICMP6	11 / 10	11 / 10	22 / 21	22 / 21

states and a 23.06% reduction in the number of state transitions. Particularly, the most significant reduction is seen in the second FSM of the OSPFv2 protocol with approximately 60% fewer nodes and 60% fewer edges.

Stage 3: Locating Implementation Differences. For each FSM difference we generated, we manually identify whether it is a true difference, i.e., a real semantic difference. We consider other differences as false positives, which are due to some inaccuracies of our tool, like the inherent limitation of pointer analysis and loop analysis. Additionally, for each true difference, we identify whether it is a bug, or it is due to implementation options allowed by protocol specifications. As shown in Table 4(a), ParDiff successfully identified 41 unique bugs in 14 network protocols, with a precision of 92.8%. For each identified bug, we generated either a bug report or a fix patch. Till submission, 25 of these bugs have been confirmed by the developers. Notably, we have patched 11 of these bugs, which have already been merged or approved by the developers in the corresponding open-source repository.

Each FSM difference is made up of several atomic branching conditions (see Definition 1). We record the number of atomic branch conditions in each difference and show the result in the last column of Table 4(a). Among the 14 protocols, there are 827 atomic conditions within 264 differences. As such, developers need to examine 3.13 atomic conditions on average for every difference detected. Since each atomic condition corresponds to one source code line, developers would examine approximately 3.13 source lines per difference (assuming they are familiar with the protocols).

5.3 RQ2: Efficiency of Each Stage in ParDiff

We executed ParDiff on all protocols in our dataset and recorded the total analysis time. Additionally, we measured the time ParDiff took for each of the three stages separately. As presented in Figure 9, ParDiff can complete the analysis in an average of 50.48 secs, with 74.15% of its time extracting formats from the source code, around 22.37% generating FSM, and only 1.97% conducting the differentiation. It supports that by applying path constraint reduction in stage 1 and FSM simplification in stage 2, ParDiff can differentiate formats and generate differences very quickly.

5.4 RQ3-1: Comparing ParDiff to Coarse-grained use of SMT Solver

As shown in Table 3, for each protocol, the SMT Solver determines the two constraints, referred to as ϕ_1 and ϕ_2, are not equivalent. However, comparing these constraints directly merely confirms the

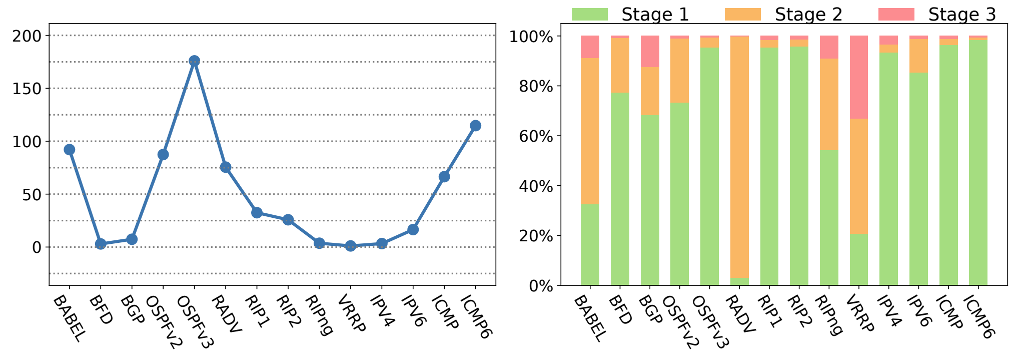


Fig. 9. Time cost (seconds) and its decomposition.

Table 3. Compare with coarse-grained use of SMT Solver. (Treat all constraints as a whole formula to query.)

Protocols	Equivalence Checking	#Atomic Cond. 1	#Atomic Cond. 2
BABEL	Not equal	296	648
BFD	Not equal	14	26
BGP	Not equal	49	93
OSPF2	Not equal	166	305
OSPF3	Not equal	93	134
RADV	Not equal	111	6
RIP1	Not equal	52	22
RIP2	Not equal	34	16
RIPng	Not equal	166	28
VRRP	Not equal	21	10
IPv4	Not equal	11	15
IPv6	Not equal	3	97
ICMP	Not equal	11	55
ICMP6	Not equal	20	44

presence of at least one difference between the two implementations. To precisely locate the origins of these discrepancies or potential bugs, we must delve deeper and inspect each individual atomic condition within both ϕ_1 and ϕ_2. The count of these atomic conditions for each implementation is recorded in Table 3. Across the 14 protocols, there are a total of 2515 atomic conditions for 14 identified differences. Hence, developers need to carefully check approximately 179.6 lines per difference to pinpoint potential bugs. This number significantly exceeds the number of 3.13 lines per difference provided by our tool, representing a significant challenge that requires nontrivial human effort.

5.5 RQ3-2: Comparing ParDiff to DPIFuzz

We compare the two tools from two perspectives: the number of discovered bugs and the efficiency of bug finding.

Bugs Identified. As shown in Table 4, our tool detects a total of 41 bugs, 40 of which are found in the first ten network protocols, while DPIFuzz can only detect 25 bugs (detected by at least one of the three runs). We observe that DPIFuzz has difficulty finding bugs in long program paths due to the difficulty of generating inputs that satisfy all the complex constraints in deep paths. It is worth mentioning that, as shown in Table 5, bugs with ID#42 to 45 can be triggered by at least one run of DPIFuzz, but are not detected by our tool. This is because, in the current implementation, if our tool detects a difference on a path, it stops bisimulation on that path. Hence, we miss the

Table 4. Precision and # Bugs Detected.

(a) **ParDiff**

Protocols	**#Diff**	**True Diff**		**FP**	**#Bug**	**#Atomic Cond.**
		#Diff. Bug	#Diff. Opt.			
BABEL	81	37	39	5 (6.1%)	18	295
BFD	2	2	0	0 (0.0%)	1	2
BGP	27	18	0	9 (33.3%)	3	75
RADV	4	2	2	0 (0.0%)	2	4
RIP1	19	11	8	0 (0.0%)	3	32
RIP2	14	6	8	0 (0.0%)	2	14
RIPng	16	12	4	0 (0.0%)	2	19
OSPFv2	25	23	0	2 (8.0%)	4	79
OSPFv3	26	3	23	0 (0.0%)	1	161
VRRP	12	11	0	1 (8.3%)	4	58
IPV4	8	0	6	2 (25.0%)	0	24
IPV6	7	7	0	0 (0.0%)	1	13
ICMP	12	0	12	0 (0.0%)	0	21
ICMP6	11	0	11	0 (0.0%)	0	30
Total	264	132	113	19 (7.2%)	41	827

(b) **DPIFuzz**

Protocols	**RUN with Equal Time**		**RUN until Coverage**	
	#Input	#Bug	#Input	#Bug
BABEL	3.50 ± 1.28	0.00 ± 0.00	49.33 ± 42.19	5.00 ± 4.08
BFD	0.50 ± 0.50	0.20 ± 0.40	9.00 ± 2.83	1.00 ± 0.00
BGP	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
OSPFv2	0.00 ± 0.00	0.00 ± 0.00	5.67 ± 4.50	1.00 ± 0.00
OSPFv3	0.00 ± 0.00	0.00 ± 0.00	5.33 ± 2.62	1.00 ± 0.00
RADV	0.40 ± 0.49	0.40 ± 0.49	0.67 ± 0.47	0.67 ± 0.47
RIP1	1.00 ± 0.77	0.20 ± 0.40	23.00± 0.82	2.00 ± 0.00
RIP2	0.00 ± 0.00	0.00 ± 0.00	45.00± 12.98	2.33 ± 0.94
RIPng	0.00 ± 0.00	0.00 ± 0.00	111.67± 12.40	3.00 ± 0.00
VRRP	0.00 ± 0.00	0.00 ± 0.00	3.00 ± 0.00	2.00 ± 0.00

opportunity of finding other bugs on that path after that difference. This could be solved by forcing the tool to continue bisimulation along the path. We put this into our future work. Additionally, our tool exhibits higher stability in bug detection. In contrast, the bugs DPIFuzz can trigger are quite random, only 12 of the 25 bugs are detected in all three runs.

Besides, we record the time each bug is detected by ParDiff and DPIFuzz. Particularly, we record the time taken by DPIFuzz to trigger the bugs in each of its three runs. We use T/O to indicate timeout, which means this bug isn't detected in an execution. The result shows that DPIFuzz spends much longer time than ParDiff to detect each bug. Besides, due to the innate randomness in fuzzing tools, DPIFuzz cannot stably detect most bugs.

Efficiency. We compare the efficiency of DPIFuzz with ParDiff in two settings: running DPIFuzz with the same time budget as ParDiff and running DPIFuzz for 24 hours.

We first study the capability of finding bugs given equal time to both tools. For each protocol, we execute DPIFuzz ten times, matching the total run time of ParDiff for each iteration. We keep track of the number of inputs and the bugs identified in every individual run. The mean and standard deviation of the results are shown in the left half part of Table 4(b), which indicates that ParDiff can hardly find any bug given the same time as our tool. Please note that we do realize this is due to the different nature of the tools and these numbers are for reference only.

```
 1   static int parse_hello_subtlv(const
 2        unsigned char *a, int alen, …)
 3   {
 4     int i = 0;
 5     while (i < alen) {
 6 -     type = a[0];
 7 +     type = a[i];
 8       if (type ==0) {
 9          i++; continue;
10       }
11       …
12     }
13   }
```

(a) Case 1

```
 1   void parse_packet(const unsigned char *packet,
 2                     int packetlen, …)
 3   {
 4     char *message = packet + 4 + i;
 5     type = message[0];
 6     if (type == MESSAGE_REQUEST) {
 7       …
 8 +     int rc = parse_request_subtlv(message[2], …);
 9 +     if (rc<0)
10 +        goto done;
11     }
12   }
13 + static int parse_request_subtlv(…){…};
```

(b) Case 2

Fig. 10. Case studies. (a) Incorrect parse sub-TLVs in BABEL. (b) Miss handling sub-TLVs in MESSAGE REQUEST TLVs.

Second, for each protocol, we run DPIFuzz for 24 hours and repeat three times. In each execution, we record the number of inputs and bugs. The mean and standard deviation of three runs are shown in the right half of Table 4(b). For each bug, we record the precise time when DPIFuzz first produces an input associated with this bug. Some bugs can be found in some runs but not in others, so we record the bug trigger time for all three runs. The average bug triggering time for DPIFuzz is 2667s, however, ParDiff only needs an average of 49s to detect each bug. Obviously, it takes a much longer time for DPIFuzz to trigger bugs than ParDiff.

5.6 RQ4: Root Cause of Discovered Bugs

We classify the 41 bugs discovered by ParDiff into three categories based on their root causes.

Category 1: Incorrect bound checks on specific fields (25 / 41). Bugs in this category imply that developers either miss checks or use incorrect checks on certain packet fields. The motivating example depicted in Figure 1 falls within this category. Such bugs can potentially lead to memory issues, delays in discarding malicious packets, and even more serious security risks.

Category 2: Incorrect parsing on specific packet types (9 / 41). This category refers to bugs when developers incorrectly handle the parsing of specific packet types. These bugs may arise from misunderstandings of the protocol specifications or coding errors in implementations. The consequences of such bugs can range from minor inconsistencies in protocol behavior to severe issues affecting the stability of the network. The following example illustrates a bug within this category:

Figure 10(a) shows a bug (ID#1 in Table 5) discovered by our tool in project BABEL. The function in the figure parses the sub-TLVs in the *hello* packet. The number of sub-TLVs is determined by *alen*, requiring a loop to visit each sub-TLV sequentially. However, in line 5, the variable *type* is always assigned with the type of the first sub-TLV, resulting in the parsing of all subsequent sub-TLVs that are based on the type of the first one. Consequently, the original implementation incorrectly parses all sub-TLVs following the first one. In the same source code file, we have found four additional similar issues and reported all these bugs to developers. Currently, all these bugs have been confirmed and resolved by the developers.

Category 3: Incomplete parsing on specific packet types (7 / 41). This category refers to the bugs when developers miss parsing some portions of certain packet types. Such bugs may arise due to an incomplete or incorrect understanding of the protocol specifications, compatibility issues stemming from protocol updates, or simply as a result of oversight during implementation. The following example falls within this category:

Table 5. Network protocol bug list. We use T/O to indicate timeout, and N/A for not applicable.

ID	Proto.	Root Causes	ParDiff	DPIFuzz	Status	Time.	First Time. DPIFuzz		
						ParDiff	1st RUN	2nd RUN	3rd RUN
1	BABEL	Incorrect parse hello sub-TLVs.	✓	✓	Confirmed	92s	1907s	13796s	T/O
2	BABEL	Incorrect parse ihu sub-TLVs.	✓	✓	Confirmed	92s	5972s	T/O	T/O
3	BABEL	Incorrect parse request sub-TLVs.	✓	×	Confirmed	92s	T/O	T/O	T/O
4	BABEL	Incorrect parse seq_req sub-TLVs.	✓	×	Confirmed	92s	T/O	T/O	T/O
5	BABEL	Incorrect parse other sub-TLVs.	✓	×	Confirmed	92s	T/O	T/O	T/O
6	BABEL	Incorrect ignoring nonzero flag bits.	✓	✓	PR merged	92s	5008s	T/O	T/O
7	BABEL	Miss handle ack_req sub-TLVs.	✓	✓	PR approved	92s	465s	8956s	T/O
8	BABEL	Miss handle ack sub-TLVs.	✓	✓	PR approved	92s	9052s	T/O	T/O
9	BABEL	Miss handle router_ID sub-TLVs.	✓	✓	PR approved	92s	8433s	T/O	T/O
10	BABEL	Miss handle NH sub-TLVs.	✓	✓	PR approved	92s	2209s	12016s	T/O
11	BABEL	Miss handle seq_req sub-TLVs.	✓	✓	PR approved	92s	1008s	T/O	T/O
12	BABEL	Miss handle request sub-TLVs.	✓	✓	PR merged	92s	1033s	9889s	T/O
13	BABEL	Incorrect router-id checking.	✓	×	Confirmed	92s	T/O	T/O	T/O
14	BABEL	Incorrect checking in route update.	✓	×	Confirmed	92s	T/O	T/O	T/O
15	BABEL	Incorrect handle route retractions.	✓	×	Confirmed	92s	T/O	T/O	T/O
16	BABEL	Miss non-zero AE check in NH.	✓	✓	PR merged	92s	2204s	4140s	T/O
17	BABEL	Miss non-zero AE check in Seq_Req.	✓	×	PR merged	92s	T/O	T/O	T/O
18	BABEL	Incorrect length check.	✓	×	Confirmed	92s	T/O	T/O	T/O
19	BFD	Miss bug version check.	✓	×	PR merged	2s	T/O	T/O	T/O
20	BGP	Miss length check in notification.	✓	×	Confirmed	7s	T/O	T/O	T/O
21	BGP	Miss asn check in open type.	✓	×	Reported	7s	T/O	T/O	T/O
22	BGP	Unhandle field length extension.	✓	×	Reported	7s	T/O	T/O	T/O
23	RADV	Miss checkings of Hop Limit.	✓	×	Reported	75s	T/O	T/O	T/O
24	RADV	Miss checkings of received RAs.	✓	✓	Reported	75s	10s	T/O	56s
25	OSPFv2	Miss non-zero check of router-id.	✓	×	Reported	87s	T/O	T/O	T/O
26	OSPFv2	Miss length check in type2.	✓	×	Reported	87s	T/O	T/O	T/O
27	OSPFv2	Miss length check in type3.	✓	×	Reported	87s	T/O	T/O	T/O
28	OSPFv2	Miss check lsa header length.	✓	✓	Reported	87s	1334s	2003s	679s
29	OSPFv3	Miss non-zero check of router-id.	✓	×	Reported	73s	T/O	T/O	T/O
30	RIP1	Miss non-zero field check.	✓	✓	PR approved	32s	912s	1470s	1058s
31	RIP1	Not handle multiple rte.	✓	✓	Confirmed	32s	194s	31s	16s
32	RIP1	Incomplete destination check.	✓	×	Reported	32s	T/O	T/O	T/O
33	RIP2	Not handle multiple rte.	✓	✓	Confirmed	26s	1453s	1560s	2463s
34	RIP2	Incomplete destination check.	✓	✓	Reported	26s	2706s	22069s	5956s
35	RIPng	Not handle multiple rte.	✓	✓	Confirmed	4s	46s	570s	292s
36	RIPng	Incomplete destination check.	✓	✓	Reported	4s	1515s	717s	733s
37	VRRP	Miss type check.	✓	×	Reported	1s	T/O	T/O	T/O
38	VRRP	Incomplete payload length check.	✓	×	Reported	1s	T/O	T/O	T/O
39	VRRP	Miss check read length.	✓	✓	Reported	1s	10s	36s	47s
40	VRRP	Buffer over-read.	✓	✓	Reported	1s	92s	108s	26s
41	IPV6	Incorrect length check.	✓	×	Reported	17s	N/A	N/A	N/A
42	BFD	Miss handle authentication.	×	✓	Confirmed	T/O	18s	11s	15s
43	OSPFv3	Miss check lsa header length.	×	✓	Reported	T/O	841s	826s	668s
43	RIP2	Incorrect subnet mask check.	×	✓	Reported	T/O	3176s	T/O	T/O
44	RIP2	Metric range check.	×	✓	Reported	T/O	T/O	4686s	T/O
45	RIPng	Incorrect nexthop rte check.	×	✓	Reported	T/O	721s	712s	744s

Figure 10(b) shows part of a pull request (ID#12 in Table 5) we submitted to fix a bug in FRRouting's BABEL implementation. The buggy version misses implementing some constraint checks while dealing with MESSAGE_REQUEST packets, leading to an incomplete input validation checking. This pull request has already been merged by the developers.

6 RELATED WORK

Differential Symbolic Analysis. Symbolic-execution-based differential testing tools primarily focus on syntactically similar programs (e.g., programs evolved from the same base version). These tools often rely on matching code snippets and data structures to reduce the symbolic execution overhead. They identify similar or divergent code segments through static analysis [Person et al. 2008, 2011; Rutledge and Orso 2022], branch divergence during symbolic execution [Cadar and Palikareva 2014; Palikareva et al. 2016], or CFG patterns [Malík and Vojnar 2021]. However, these tools can struggle if programs from independent projects are substantially different in their syntactic structures. Contrarily, ParDiff provides a complementary approach for analyzing different implementations, even when their syntactic structures vary significantly.

Existing differential model checking techniques [Ferreira et al. 2021; Fiterău-Broştean et al. 2016]are mainly used to compare high-level state-machine representations of protocols. These are effective in pinpointing operational differences between different protocol versions or implementations. For instance, Prognosis [Ferreira et al. 2021] models state transitions and message exchange patterns (e.g. TCP 3-way handshake) of TCP, as well as QUIC's flow control mechanisms, for comparison. While these methods are well-suited for studying the overall logic of protocol implementations, they might not thoroughly check the robustness of individual parsers when faced with malformed or unexpected input.

Fuzzing and Differential Fuzzing. Traditional fuzzing tools, like BooFuzz [Pereyda 2023], primarily aim to uncover crashes or assertion failures, rather than semantic bugs. Differential fuzzing explores potential behavior divergence between two programs and is capable of detecting semantic bugs. However, these techniques [Yang et al. 2021; Zou et al. 2021] encounter several challenges. Firstly, their effectiveness strongly depends on the quality of the inputs and may suffer from low coverage of code. Secondly, substantial human effort is needed to identify and locate bugs from a large number of inputs. Lastly, these techniques tend to be slow, often taking several hours or days to converge. To mitigate these challenges, we propose ParDiff, a high-coverage bug detection technique with efficient bug identification and localization.

Hybrid Techniques. Symbolic execution and fuzzing can be combined to identify semantic differences between two program versions. HyDiff [Noller et al. 2020] leverages both dynamic symbolic execution and concolic testing to find regression bugs. These tools also suffer from the limitations of applying differential symbolic execution in syntactically disparate implementations.

Input Grammar Synthesis. Input grammar synthesis techniques [Bastani et al. 2017; Gopinath et al. 2020; Lin et al. 2010] are widely used to generate grammar describing the expected syntactic structure of program inputs. These grammars help ensure that inputs adhere to specified formats and can be valuable for generating test inputs. However, it is worth noting that these techniques often suffer from performance issues [Bendrissou et al. 2022]. Furthermore, input grammar synthesis techniques typically target context-free grammars, which imposes certain limitations on their applicability. In contrast, protocol formats include semantic constraints among different bytes in a protocol message. These constraints may involve arithmetic operations, bit-level manipulations, or context-dependent rules that go beyond the capabilities of simple context-free grammars.

7 CONCLUSION

This work proposes ParDiff to detect bugs hidden in network protocol parsers, which could hardly be detected by conventional tools. ParDiff extracts normalized protocol formats as finite state machines from different implementations of the same protocol, and leverages differential analysis to locate bugs in the code. ParDiff successfully detects 41 semantic bugs, with 25 confirmed or fixed.

ACKNOWLEDGMENTS

We thank Benjamin Delaware and Zhuo Zhang for insightful discussions. We also thank the anonymous reviewers for their comments and suggestions. This work was supported in part by DARPA VSPELLS HR001120S0058, NSF1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

Fernando Arnaboldi. 2023. XDiFF. https://github.com/IOActive/XDiFF.

Domagoj Babic and Alan J. Hu. 2008. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08).* ACM, 211–220. https://doi.org/10.1145/1368088.1368118

Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. ACM, 13–24. https://doi.org/10.1145/3368089.3409757

Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. 2011. A decade of software model checking with SLAM. *Commun. ACM* 54, 7 (2011), 68–76. https://doi.org/10.1145/1965724.1965743

Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P Sadayappan. 2016. PolyCheck: dynamic verification of iteration space transformations on affine programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, 539–554. https://doi.org/10.1145/2837614.2837656

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, 95–110. https://doi.org/10.1145/3062341.3062349

Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. 2022. "Synthesizing Input Grammars": A Replication Study. In *Proceedings of the th International Conference on Software Engineering (PLDI '22)*. ACM, 260–268. https://doi.org/10.1145/3519939.3523716

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2009. Bounded Model Checking. In *Handbook of Satisfiability*, Vol. 185. IOS Press, 457–481. https://doi.org/10.3233/978-1-58603-929-5-457

Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS '09)*. ACM, 621–634. https://doi.org/10.1145/1653662.1653737

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. USENIX, 209–224. https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems

Cristian Cadar and Hristina Palikareva. 2014. Shadow symbolic execution for better testing of evolving software. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion '14)*. ACM, 432–435. https://doi.org/10.1145/2591062.2591104

Chia Yuan Cho, Vijay D'Silva, and Dawn Song. 2013. BLITZ: Compositional bounded model checking for real-world programs. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*. IEEE, 136–146. https://doi.org/10.1109/ASE.2013.6693074

Juliusz Chroboczek. 2023. parse_update_subtlv in Jech. https://github.com/jech/babeld/blob/babeld-1.12-branch/message.c.

Juliusz Chroboczek and David Schinazi. 2023. RFC 8966: The Babel Routing Protocol. https://www.rfc-editor.org/rfc/rfc8966.html.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. ACM, 1027–1040. https://doi.org/10.1145/3314221.3314596

FRR community. 2023. The FRRouting protocol suite. https://github.com/FRRouting/frr.

Wikipedia contributors. 2022. List of open-source routing platforms. https://wikipedia.org/wiki/List_of_open-source_routing_platforms.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

FRR Developers. 2023. FRRouting. https://github.com/FRRouting/frr/blob/ab68283ceedc05ea1a7f9c54f03a87f5dc199a01/babeld/message.c.

Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. ACM, 762–774. https://doi.org/10.1145/3452296.3472938

Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *Computer Aided Verification (CAV '16, Vol. 9780)*. Springer, 454–471. https://doi.org/10.1007/978-3-319-41540-6_25

Raffaella Gentilini, Carla Piazza, and Alberto Policriti. 2003. From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning* 31, 1 (2003), 73–103. https://doi.org/10.1023/A:1027328830731

Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44. https://doi.org/10.1145/2093548.2093564

Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. ACM, 172–183. https://doi.org/10.1145/3368089.3409679

Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *22nd USENIX Security Symposium (USENIX Security '13)*. USENIX, 49–64. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/haller

Heartbleed. 2020. The Heartbleed Bug. https://heartbleed.com.

Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (S&P '20)*. IEEE, 1613–1627. https://doi.org/10.1109/SP40000.2020.00063

Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. 2011. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *2011 IEEE Symposium on Security and Privacy (S&P '11)*. IEEE, 347–362. https://doi.org/10.1109/SP.2011.41

Bakhadyr Khoussainov and Anil Nerode. 2012. *Automata theory and its applications*. Vol. 21. Springer. https://doi.org/10.1007/978-1-4612-0171-7

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification (CAV '12, Vol. 7358)*. Springer, 712–717. https://doi.org/10.1007/978-3-642-31424-7_54

Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential Assertion Checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, 345–355. https://doi.org/10.1145/2491411.2491452

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*. IEEE, 75. https://doi.org/10.1109/CGO.2004.1281665

Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. *IEEE Transactions on Software Engineering* 36, 05 (2010), 688–703. https://doi.org/10.1145/1453101.1453114

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundness: a manifesto. *Commun. ACM* 58, 2, 44–46. https://doi.org/10.1145/2644805

Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. ACM, 175–186. https://doi.org/10.1145/3236024.3236082

Viktor Malík and Tomáš Vojnar. 2021. Automatically checking semantic equivalence between versions of large-scale C projects. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST '21)*. IEEE, 329–339. https://doi.org/10.1109/ICST49551.2021.00045

Mares Martin, Machek Pavel, Filip Ondrej, and CZ.NIC. 2023. BIRD internet routing daemon. https://gitlab.nic.cz/labs/bird.

Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-specific equivalence checking. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. ACM, 441–451. https://doi.org/10.1145/3238147.3238178

Madanlal Musuvathi and Dawson R. Engler. 2004. Model Checking Large Network Protocol Implementations. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*. USENIX, 12. https://doi.org/10.1145/3092282.3092289

Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: Hybrid differential software analysis. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. ACM, 1273–1285. https://doi.org/10.1145/3377811.3380363

Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 1181–1192. https://doi.org/10.1145/2884781.2884845

Joshua Pereyda. 2023. BooFuzz. https://github.com/jtpereyda/boofuzz.

Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Pundefinedsundefinedreanu. 2008. Differential Symbolic Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE '16)*. ACM, 226–237. https://doi.org/10.1145/1453101.1453131

Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed Incremental Symbolic Execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, 504–515. https://doi.org/10.1145/1993498.1993558

Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (S&P '17)*. IEEE, 615–632. https://doi.org/10.1109/SP.2017.27

David A. Ramos and Dawson R. Engler. 2011. Practical, low-effort equivalence verification of real code. In *Computer Aided Verification (CAV '11, Vol. 6806)*. Springer, 669–685. https://doi.org/10.1007/978-3-642-22110-1_55

David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium (USENIX Security '15)*. USENIX, 49–64. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

Gaganjeet Singh Reen and Christian Rossow. 2020. DPIFuzz: a differential fuzzing framework to detect DPI elusion strategies for QUIC. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC '20)*. ACM, 332–344. https://doi.org/10.1145/3427228.3427662

Richard Rutledge and Alessandro Orso. 2022. Automating Differential Testing with Overapproximate Symbolic Execution. In *2022 15th IEEE Conference on Software Testing, Verification and Validation (ICST '22)*. IEEE, 256–266. https://doi.org/10.1109/ICST53961.2022.00035

Davide Sangiorgi. 1998. On the bisimulation proof method. *Mathematical Structures in Computer Science* 8 (1998), 447 – 479. https://api.semanticscholar.org/CorpusID:14986397

Qingkai Shi, Junyang Shao, Yapeng Ye, Mingwei Zheng, and Xiangyu Zhang. 2023. Lifting Network Protocol Implementation to Precise Format Specification with Security Applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. ACM, 1287–1301. https://doi.org/10.1145/3576915.3616614

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 693–706. https://doi.org/10.1145/3192366.3192418

Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. 2011. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *Proceedings of the 9th Asian Symposium on Programming Languages and Systems (APLAS '11)*. Springer, 155–171. https://doi.org/10.1007/978-3-642-25318-8_14

Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems* 34, 3 (2012), 1–35. https://doi.org/10.1145/2362389.2362390

Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bracevac, and Tiark Rompf. 2023. Compiling Parallel Symbolic Execution with Continuations. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '23)*. IEEE, 1316–1328. https://doi.org/10.1109/ICSE48619.2023.00116

Yichen Xie and Alex Aiken. 2005. Scalable error detection using Boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, 351–363. https://doi.org/10.1145/1047659.1040334

Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*. USENIX, 349–365. https://www.usenix.org/conference/osdi21/presentation/yang

Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing. In *USENIX Annual Technical Conference (ATC '21)*. USENIX, 489–502. https://www.usenix.org/conference/atc21/presentation/zou

Received 21-OCT-2023; accepted 2024-02-24