Differential Testing of Bundle Protocol v7 Implementations: A Preliminary Report

Stephan Havermans*†, Lars Baumgärtner‡, Marcus Wallum‡ and Juan Caballero*
*IMDEA Software Institute
†Universidad Politécnica de Madrid
‡European Space Agency

***Abstract*—Communication protocols are integral to space systems. As such, their implementations should be thoroughly tested for conformance, interoperability, and security issues. This work-in-progress presents a novel differential approach for protocol testing and preliminary results from its application. Our approach takes as input multiple implementations of the same protocol, executes the implementations on a large number of inputs, and identifies deviation inputs that are accepted (i.e., successfully parsed and validated) by some implementations but rejected by other implementations. We have applied our approach to test three implementations of Bundle Protocol version 7, identifying 51,373 deviation inputs grouped into 432 clusters. So far, we have reported 14 manually verified conformance issues and vulnerabilities. Ten have already been fixed.**

I. Introduction

Robust communication is vital to the operation of space systems. As such, protocol implementations should be thoroughly tested to ensure that they function as described in their specification (i.e., *protocol conformance testing* [1], [2]) and that they properly communicate with other implementations of the same protocol (i.e., *interoperability testing* [3], [4]). Such testing is fundamental because communication protocols can be complex containing many mandatory and optional features. Furthermore, protocol specifications are written in accessible but informal text. Despite guidelines on how to write specifications [5]–[7] and much effort by working groups to make specifications clear, ambiguity can still creep in, leading to diverging interpretations by different developers [8].

Protocol conformance and interoperability testing are frequent activities. In fact, both the Internet Standards Process [3] and the Committee for Space Data Systems (CCSDS) [4] require two independent and interoperable implementations for a protocol specification to become a full standard. Currently, the CCSDS is performing interoperability tests of the Bundle Protocol version 7 (BPv7) [9], a store-and-forward protocol designed for Delay/Disruption-Tolerant Networking (DTN) across networks with intermittent connectivity and long delays such as those in space communication. BPv7 is a critical part of initiatives to build communication networks beyond Earth like Moonlight [10], LunaNet [11], and the Solar System Internet [12]. Conformance and interoperability issues in forwarding protocols such as BPv7 can introduce subtle security issues. For example, BPv7 provides fragmentation and reassembly functionality that has often been leveraged for attacks against other protocols [13]–[15] and for bypassing defenses such as firewalls and intrusion detection systems [16]. Conformance issues can also be used for uniquely fingerprinting implementations [17]. Furthermore, given the intermittent communication windows and limited bandwidth of space links, conformance issues can be leveraged by attackers to exhaust resources leading to denial of service (DoS) attacks.

Protocol conformance and interoperability testing are highly manual processes. Previous works have proposed automated approaches using formal methods [18]–[20] and symbolic execution [17], [21], [22]. However, those approaches have not been largely adopted likely due to their complexity and the requirement of a formal protocol specification for formal method approaches [23]. Other works present fuzzers to identify inputs that crash network protocol implementations [24]–[27]. Fuzzing approaches are simpler since they do not require a formal specification or symbolic execution. However, protocol conformance and interoperability issues do not typically crash an implementation. Instead, they may introduce logical bugs that are hard to identify.

This work presents a novel differential approach for automated protocol testing that leverages the simplicity and efficiency of state-of-the-art fuzzing approaches. Our approach takes as input multiple implementations of the same protocol and automatically identifies *deviations*, that is, inputs that are accepted (i.e., successfully parsed and validated) by some implementations but are rejected by other implementations. The identified deviations are typically caused by at least one implementation violating the specification, i.e., accepting invalid inputs or rejecting valid inputs.

The problem of finding deviations among implementations of the same network protocol was first introduced by Brumley et al. [17]. The key difference is that our approach avoids the use of symbolic execution. Instead, our approach leverages input generation techniques used by fuzzers (e.g., mutational [28], coverage-guided [29], grammar-based [30]). Since our approach does not use symbolic execution, it is faster and simpler to use. It does not require access to the source code and can be applied to implementations written in different programming languages.

We apply our approach to the latest version (at the time of testing) of three implementations of BPv7, detailed in Table I. Two of these implementations (ESA BP and µD3TN) are part of current BPv7 interoperability tests. One is developed by the European Space Agency (ESA) in Java and the other by

TABLE I
BPv7 IMPLEMENTATIONS TESTED.

Program	Org.	Lang.	SLOC	Public	Version
ESA BP	ESA	Java	60K	✗	448f180
µD3TN [33]	D3TN	C	51K	✓	0.14.2
bp7-rs [34]	OSS	Rust	4.2K	✓	0.10.7

a company in C. The third one (bp7-rs) is an open-source project written in Rust [31], which we select to illustrate that Rust's memory-safety properties do not protect against conformance issues. BPv7 poses several challenges. First, by default, BPv7 messages (called *bundles*) do not generate a response. This complicates observing whether an input was accepted or rejected by an implementation. To address this, we build *error checkers* that apply regular expressions to the content of the produced logs. Furthermore, BPv7 encodes fields using the Concise Binary Object Representation (CBOR) [32]. This complicates input generation as valid bundles need to be properly CBOR-encoded.

We have applied our approach to generate 15M inputs using three input generation strategies (mutational, coverage-guided, and grammar-based), identifying 51,373 deviation inputs grouped into 432 clusters. Until now, we have manually examined a small subset of the clusters. Some clusters capture DoS vulnerabilities triggered by malformed bundles. In most other examined clusters, two implementations correctly followed the specification while a third did not. However, in three clusters, the majority was incorrect, so we had to file two bug reports for the same cluster. So far, we have reported 14 bugs, 10 of which have already been fixed.

II. Approach

Figure 1 summarizes our approach. It takes as input multiple (at least two) implementations of the same protocol. It outputs deviations, i.e., inputs that are accepted by some of the implementations but rejected by others, which are grouped in clusters with the same underlying cause.

In detail, our approach comprises four modules: *input generator*, *executor*, *deviation detector*, and *clustering*. The input generator produces a large number of concrete inputs corresponding to both valid and invalid bundles. It implements three popular input generation strategies: mutational [28], coverage-guided [29], and grammar-based [30]. The generated inputs are stored to form an *input corpus* for each strategy. The executor runs all implementations on the produced inputs and collects the generated logs. For each implementation, we build an *error checker* that examines its logs using regular expressions to determine whether the processed input was accepted or rejected. The deviation detector applies the error checker to each log. If all implementations accept or reject the input, the input is discarded as it does not trigger a deviation. If at least one implementation disagrees with the rest (e.g., two implementations accept the input and one rejects it) then the deviation input and the logs are stored.

After all generated inputs have been processed, the clustering module groups the identified deviation inputs having the same underlying root cause. For this, it leverages the stored logs, which often detail why an input was rejected.

A. Input Generator

The goal of the input generator is to produce large numbers of inputs. The generated inputs may be valid or invalid messages according to the protocol specification. Generating both valid and invalid inputs is important to identify both implementations that are too strict, i.e., may reject inputs that the specification states are allowed, and implementations that are too lax, i.e., may accept inputs that the specification states should not be allowed. The module provides three input generation strategies leveraging two popular fuzzers. It uses AFL++ [35] for mutational and coverage-guided input generation and Peach [36] for grammar-based input generation.

Mutational. This strategy takes a set of *seed inputs* and applies mutations to those seeds to generate new inputs, e.g., flipping bits, replacing bytes with interesting edge cases (e.g., zero, 0xFF), or combining two inputs. To implement this strategy we run AFL++ in non-instrumented mode and build a custom mutator that receives the generated input after the AFL++ default mutators have been applied, writes it to file, and stops the input generation process after a configurable maximum number of inputs is reached. We feed AFL++ with three seed BPv7 bundles, each taken from the documentation of one of the implementations.

Coverage-guided. One implementation is instrumented to track which parts of the code are executed on a given input. When a mutated input triggers new coverage, i.e., executes new code paths, the input is saved and prioritized as a basis for further mutations. To implement this strategy we use our custom AFL++ mutator but with coverage tracking enabled. We use the same three seeds as in the mutational strategy. Since the instrumentation of AFL++ is largely designed for C/C++ programs, we apply AFL++ on µD3TN, which is written in C. The executor will run all three implementations on the generated inputs. While the produced inputs maximize coverage on µD3TN, we still expect them to execute multiple branches on the other two implementations. For example, we expect that some (but not all) of the µD3TN validation checks may be shared with the other implementations.

Grammar-based. This strategy produces inputs using a pre-defined protocol grammar. The grammar allows generating syntactically valid inputs that are more likely to pass initial parsing stages and reach deeper logic in the implementations. Unfortunately, a protocol grammar is typically not available. However, in this case, we could leverage a BPv7 grammar for the Peach fuzzer (i.e., a Peach *pit*) we had developed in a recent vulnerability-finding project [37]. Peach pits include a data model describing the format of the messages and a state model describing the sequence of messages to send to the target. For each message, the data model captures the sequence of fields with their type and length, and which fields capture

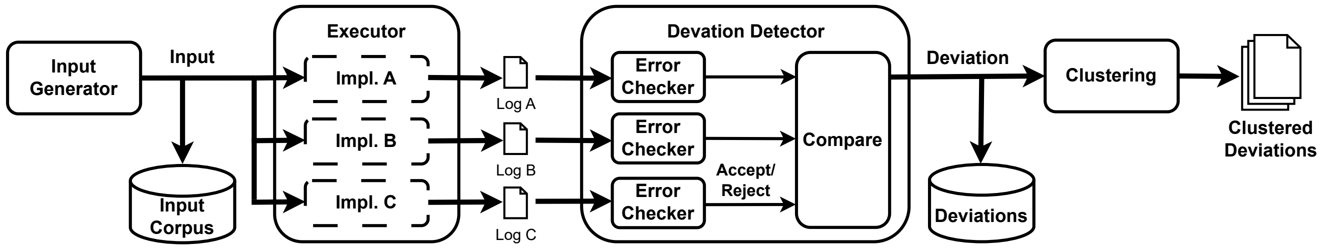

Fig. 1. Overview of the differential testing pipeline.

relations to other fields, such as length fields. To generate valid inputs, the pit is complemented with C# plugins that encode the fields using CBOR and compute the checksums. The state model simply requests the fuzzer to send the built input to the target on top of UDP. While Peach uses the grammar to generate valid protocol messages, it also applies mutators that may convert valid inputs into invalid inputs.

B. Executor

The executor runs each protocol implementation on the generated inputs and saves the logs of the execution. BPv7 implementations can be large, as illustrated in Table I. However, we expect many deviations to originate from the input processing code. Thus, as a starting point, we focus on finding deviations in the parsing stage. Both µD3TN and bp7-rs provide stand-alone executables that only implement the parsing of bundles. These executables are typically provided so that the parsing code can be easily fuzzed since parsers are known to frequently contain vulnerabilities. For these two implementations, we can simply run the stand-alone executables with the bundle (as a hexadecimal string) passed as a parameter on the command line and redirect standard output and standard error to a log file. Since the executable is re-run on each input, this guarantees all inputs are processed from the same initial state. ESA BP does not provide a stand-alone parser executable, so we implemented a Java harness that directly invokes the main parsing function with the bundle read from a file. The harness outputs a predefined success message if the parsing function successfully returns and a predefined error message if an uncaught Java exception happens.

Our goal is not to detect crashes since a differential approach is not needed for that, i.e., each implementation can be fuzzed independently. However, some generated inputs may cause the implementations to terminate unexpectedly (e.g., a segmentation fault in C/C++ programs or an unhandled exception in Java programs). If the executor detects unexpected termination, it flags the input as rejected. In Section III, we show that our approach indeed detects some denial-of-service vulnerabilities among the identified deviations.

C. Deviation Detector

To determine if an input is accepted or rejected by an implementation we need an observable property. In protocols that trigger a response, whether the input was accepted can often be determined from the response to the input, e.g., using the status code in an HTTP response [17]. However, by default BPv7 nodes do not provide a response when a bundle is received. There is an option in BPv7 to request status reports but since we focus on the parser component of the implementations status reports are not available.

To address this issue, we build an error checker for each implementation. It applies regular expressions to the content of the produced logs to identify error messages. If an error is found, the input is rejected. The absence of errors means the input was accepted. For example, in bp7-rs if the log contains sentences starting with “Error decoding bundle!”, “Error validating bundle:”, or “Unknown crc type” then the input was rejected, else it was accepted. If the input is accepted or rejected by all implementations, then it is discarded as it does not trigger a deviation. If an implementation disagrees with the others, the deviation input and the logs are saved.

D. Clustering

Multiple deviation inputs may be due to the same underlying cause, e.g., the same protocol conformance violation. We cluster the deviation inputs using a two-step approach. First, we generate a vector for each deviation with the accept/reject decisions by each implementation and group the deviations with the same vector. Since we test three implementations and each deviation requires at least one accepting and one rejecting implementation, there are six possible initial clusters.

Second, we split the initial clusters so that inputs in the same cluster produce the same normalized error messages in the implementations that reject the input. Intuitively, if an implementation reports different errors for different inputs, those inputs are unlikely to be due to the same root cause and should be in different clusters. The normalization replaces dates and numbers in the error messages with placeholders. Inputs rejected by two implementations only end up in the same cluster if both normalized error messages are the same.

III. Evaluation

We evaluate our approach on the three implementations in Table I. We first detail the deviations found in Section III-A and then discuss the identified bugs in Section III-B.

A. Deviations Found

Table II compares the efficacy of the three input generation approaches for finding deviations. We use each input generation technique to produce 1M inputs, execute the three

TABLE II
COMPARISON OF INPUT GENERATION STRATEGIES. EACH ROW PRESENTS AN AVERAGE OF 5 ROUNDS, EACH GENERATING 1M INPUTS, EXCEPT THE TWO RIGHTMOST COLUMNS THAT CAPTURE THE TOTAL NUMBER OF UNIQUE DEVIATION INPUTS AND CLUSTERS ACROSS THE 5 ROUNDS.

			Accepted Inputs				Rejected Inputs	Deviation	Total Deviations	
Strategy	**Tool**	**Runtime**	**bp7-rs**	**µD3TN**	**ESA BP**	**All**	**All**	**Inputs**	**Inputs**	**Clusters**
Mutational	AFL++	5m 27s	2.50%	2.79%	3.22%	2.44%	96.46%	1.10%	19,516	118
Coverage-guided	AFL++	1m 47s	1.29%	1.58%	1.40%	1.09%	98.06%	0.85%	28,615	313
Grammar-based	Peach	11h 55m 29s	1.22%	1.62%	1.50%	1.18%	98.11%	0.71%	4,364	149

implementations on those inputs and examine how many deviations are found. Since input generation techniques are non-deterministic, we repeat the process five times and average the results. Thus, in total, we generated 15M inputs.

The larger ratio of deviation inputs is found with the mutational strategy (1.10%) followed by coverage-guided (0.85%) and grammar-based (0.71%). The vast majority (96.46%-98.11%) of generated inputs are rejected by all three implementations (e.g., due to encoding, checksum, and logic violations), while 1.09%–2.44% inputs are accepted by the three implementations. Those inputs are discarded as they do not manifest deviations. The coverage-guided strategy generates more accepted inputs on µD3TN (1.58%) since coverage is maximized for that implementation, but the accepted inputs on the other two implementations (1.29%–1.40%) is still larger than those of the grammar-based strategy (1.22%–1.50%).

The same input may be produced by different strategies. After removing such duplicates, we are left with 51,373 unique deviation inputs, grouped into 432 clusters. The median cluster size is 3 inputs, with a mean of 118.9. Of the 432 clusters, 127 (29.4%) have a single input, 167 (38.7%) have between 2 and 10 inputs, 86 (19.9%) have between 10 and 100 inputs, 38 (8.8%) between 100 and 1K, and 14 (3.2%) have over 1K inputs.

Table III shows the split of the deviation inputs and clusters by initial cluster type. The most common case, responsible for 36.1% of deviation inputs and 68.3% of clusters, is that both bp7-rs and ESA BP reject the input but µD3TN accepts it. The three implementations first decode the received bundle using a third-party CBOR library and then validate the decoded fields. Based on the error messages, we estimate that 87% of the clusters capture deviations in the CBOR decoding (i.e., libraries disagree on whether the input is valid CBOR content) and 13% capture deviations in the validation checks.

B. Bugs Reported (So Far)

We have identified 28 clusters where the ESA BP parser throws a Java exception such as *NullPointerException* or *IndexOutOfBoundsException*. We replayed deviations in these clusters against a full ESA BP node. The parser exceptions are not caught upstream causing the main parsing thread to crash and stop parsing bundles, leading to a DoS. While the exception causes are varied, the fix is likely the same, i.e., catch all exceptions from the parser. Thus, we have filed a single bug report to ESA BP to fix these vulnerabilities.

TABLE III
SPLIT OF DEVIATION INPUTS AND CLUSTERS, A CHECK MARK (✓) INDICATES ACCEPTANCE, A CROSS (✗) REJECTION OF THE INPUT.

bp7-rs	**µD3TN**	**ESA BP**	**Deviation Inputs**	**Clusters**
✗	✓	✗	18,560 (36.1%)	295 (68.3%)
✗	✗	✓	18,745 (36.5%)	37 (8.6%)
✓	✗	✗	7,383 (14.4%)	32 (7.4%)
✗	✓	✓	4,858 (9.5%)	28 (6.5%)
✓	✓	✗	1,346 (2.6%)	37 (8.6%)
✓	✗	✓	481 (0.9%)	3 (0.7%)
All deviations			51,373 (100%)	432 (100%)

For the remaining clusters, we aim to determine which implementations are responsible for the deviations and file them a bug report. For this, we compare the error messages of the rejecting implementations with the deviation inputs and the BPv7 specification. If the error messages match the specification, then the accepting implementations are responsible. Otherwise, the rejecting implementations are responsible.

So far, we have analyzed 11 clusters. We have focused on validation deviations since CBOR-decoding deviations should be reported to the external libraries. All 11 clusters reveal protocol conformance issues. In eight clusters, one implementation violates the specification. In the other three, two implementations are at fault. Thus, the majority is often right, but not always. We have filed 14 bug reports summarized in Table IV. Of those, 10 have already been fixed.

Of the reported bugs, 11 are due to *overly lenient receivers*, i.e., implementations accepting invalid inputs. It is often mentioned that protocol implementations should be strict in what they send and lenient in what they receive [38]. However, a recent RFC argues that implementations being too lenient may affect protocol robustness [39]. One example bug already fixed was a conformance issue in µD3TN where bundles with their source node set to `dtn:none` were allowed to request status reports. The specification explicitly prohibits this, as it is not clear where the status reports should be sent. Both ESA BP and bp7-rs rejected such bundles but µD3TN accepted them.

IV. FUTURE WORK

As a work-in-progress, we plan to explore multiple issues next. First, we plan to examine the remaining clusters. Since some bugs have already been patched, we plan to re-test the patched implementations and use the results to evaluate our clustering, e.g., whether fixing one cluster removes other clusters or if fixing a cluster does not eliminate a cluster.

TABLE IV
REPORTED BUGS.

Impl.	Issue	Reason	Status
µD3TN	Accepts invalid BP version numbers	Too lenient receiver	Fixed
µD3TN	Accepts set status report flags with Source ID none	Too lenient receiver	Fixed
µD3TN	Accepts invalid payload block number	Too lenient receiver	Fixed
µD3TN	Accepts bundle w/ zero timestamp and no Bundle Age block	Too lenient receiver	Fixed
µD3TN	Accepts duplicate block numbers	Too lenient receiver	Fixed
bp7-rs	Accepts invalid block ordering	Too lenient receiver	Fixed
bp7-rs	Accepts malformed `dtn:none` EIDs	Too lenient receiver	Fixed
bp7-rs	Rejects zero timestamp w/ bundle age block present	Incorrect validation	Fixed
bp7-rs	Rejects legal processing flag combination (fragmentation flags)	Incorrect validation	Fixed
ESA BP	Accepts malformed `dtn:none` EIDs	Too lenient receiver	Reported
ESA BP	Accepts invalid CRC type value and wrong CBOR type	Too lenient receiver	Reported
ESA BP	Accepts duplicate block numbers	Too lenient receiver	Reported
ESA BP	Accepts bundle w/ zero timestamp and no Bundle Age block	Too lenient receiver	Reported
ESA BP	IndexOutOfBoundsException kills parsing thread	Uncaught exception	Fixed

We also plan to test full BPv7 nodes. This would allow us to identify deviations beyond the parsing stage and to determine if status reports can replace the log error checkers. We also plan to test other protocols to show how the approach generalizes. Finally, we plan to discuss our results with the relevant CCSDS groups and to release our implementation once the project is completed.

ACKNOWLEDGMENTS

This work was partially funded by the Spanish Government MCIN/AEI/10.13039/501100011033/ through grants PID2022-142290OB-I00 (ESPADA) and TED2021-132464B-I00 (PRODIGY). These grants are co-funded by European Union ESF, EIE, FEDER, and NextGeneration funds.

REFERENCES

[1] ISO, "ISO/IEC 9646-1:1994 Information technology — Open Systems Interconnection — Conformance testing methodology and framework," 1996.

[2] "CCSDS A20.1-Y-1: CCSDS Implementation Conformance Statements," April 2014.

[3] D. Crocker, E. Burger, and R. Housley, "Reducing the Standards Track to Two Maturity Levels," RFC 6410, Oct. 2011.

[4] CCSDS, "Organization and Processes for the Consultative Committee for Space Data Systems," April 2014.

[5] S. O. Bradner, "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, Mar. 1997.

[6] G. D. Scott, "Guide for Internet Standards Writers," RFC 2360, Jun. 1998.

[7] S. Ginoza and H. Flanagan, "RFC Style Guide," RFC 7322, Sep. 2014.

[8] J. Yen, T. Lévai, Q. Ye, X. Ren, R. Govindan, and B. Raghavan, "Semi-Automated Protocol Disambiguation and Code Generation," in *ACM SIGCOMM Conference*, 2021.

[9] S. Burleigh, K. Fall, and E. J. Birrane, "Bundle Protocol Version 7," RFC 9171, Jan. 2022.

[10] ESA, "Moonlight programme: Pioneering the path for lunar exploration," October 2024. [Online]. Available: https://www.esa.int/Applications/Connectivity_and_Secure_Communications/ESA_s_Moonlight_programme_Pioneering_the_path_for_lunar_exploration

[11] K. Schauer and D. Baird, "LunaNet: Empowering Artemis with Communications and Navigation Interoperability," Oct. 2021. [Online]. Available: https://www.nasa.gov/humans-in-space/lunanet-empowering-artemis-with-communications-and-navigation-interoperability/

[12] European Space Agency, "Extending the internet into space," 2023. [Online]. Available: https://esoc.esa.int/extending-internet-space

[13] Y. Gilad and A. Herzberg, "Fragmentation considered vulnerable," *ACM Transactions on Information and System Security*, vol. 15, no. 4, pp. 1–31, 2013.

[14] M. Vanhoef, "Fragment and forge: breaking {Wi-Fi} through frame aggregation and fragmentation," in *USENIX Security Symposium*, 2021.

[15] "pmtud is not panacea: Revisiting ip fragmentation attacks against tcp."

[16] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," Secure Networks, Inc, Tech. Rep., 1998.

[17] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation," in *USENIX Security Symposium*, 2007.

[18] A. T. Dahbura, K. K. Sabnani, and M. U. Uyar, "Formal methods for generating protocol conformance test sequences," *Proceedings of the IEEE*, vol. 78, no. 8, pp. 1317–1326, 2002.

[19] M. Musuvathi, D. R. Engler *et al.*, "Model Checking Large Network Protocol Implementations," in *USENIX Symposium on Networked Systems Design and Implementation*, 2004.

[20] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets," in *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2005.

[21] J. Song, T. Ma, C. Cadar, and P. Pietzuch, "Rule-based Verification of Network Protocol Implementations using Symbolic Execution," in *IEEE International Conference on Computer Communications and Networks*, 2011.

[22] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein, "Analyzing protocol implementations for interoperability," in *USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[23] R. Lai, "A survey of communication protocol testing," *Journal of Systems and Software*, vol. 62, no. 1, pp. 21–46, 2002.

[24] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *IEEE International Conference on Software Testing, Validation and Verification*, 2020.

[25] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *USENIX Security Symposium*, 2022.

[26] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, p. 191, 2022.

[27] N. Bars, M. Schloegel, N. Schiller, L. Bernhard, and T. Holz, "No Peer, no Cry: Network Application Fuzzing via Fault Injection," in *ACM Conference on Computer and Communications Security*, 2024.

[28] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling Black-Box Mutational Fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security*, 2013.

[29] M. Zalewski, "American fuzzy lop (afl)," http://lcamtuf.coredump.cx/afl, 2017.

[30] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[31] L. Baumgärtner, J. Höchst, and T. Meuser, "B-dtn7: Browser-based disruption-tolerant networking via bundle protocol 7," in *2019 Inter-*

national Conference on Information and Communication Technologies for Disaster Management (ICT-DM), Dec 2019.

[32] C. Bormann and P. E. Hoffman, “Concise Binary Object Representation (CBOR),” RFC 8949, Dec. 2020.

[33] “ud3tn,” 2025. [Online]. Available: https://gitlab.com/d3tn/ud3tn

[34] “dtn7-rs: An implementation of the bundle protocol in rust,” 2025. [Online]. Available: https://github.com/dtn7/dtn7-rs

[35] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in *USENIX Workshop on Offensive Technologies*, 2020.

[36] “Gitlab protocol fuzzer community edition,” https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce, 2024.

[37] “Anonymized for submission,” 2025.

[38] J. Postel, “Internet Protocol,” RFC 791, Sep. 1981.

[39] M. Thomson and D. Schinazi, “Maintaining Robust Protocols,” RFC 9413, Jun. 2023.