Fuzzing Space Communication Protocols

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

***Abstract*—Space systems are critical assets and protecting them against cyberattacks is a paramount challenge that has received limited attention. In particular, it is fundamental to secure spacecraft communications by identifying and removing potential vulnerabilities in the implementations of space (communication) protocols, which could be remotely exploited by attackers. This work reports our preliminary experiences when fuzzing five open-source implementations of four space protocols using two approaches: grammar-based fuzzing and coverage-guided fuzzing. To enable the fuzzing, we created grammars for the protocols and custom harnesses for the targets. Our fuzzing identified 11 vulnerabilities across four targets caused by typical memory-related bugs such as double-frees, out-of-bounds reads, and the use of uninitialized variables. We responsibly disclosed the vulnerabilities. To date, 5 vulnerabilities have been patched and 4 have been awarded CVE identifiers. Additionally, we discovered a discrepancy in how one target interprets a protocol standard, which we reported and has since been fixed.**

I. Introduction

Space systems are integral to modern society, facilitating a vast array of critical services such as global communications, navigation, weather forecasting, environmental monitoring, and national security. Given their far-reaching impact and cost, protecting space systems is paramount. Historically, cyberattacks on space systems were a secondary concern, focusing on achieving mission objectives [33]. However, an increasing number of studies emphasize that protecting space systems against cyberattacks should become a primary concern [26], [44], [45]. Those works highlight an increasing attack surface, for example as a result of exponential launch cadences and interest in reducing development costs by using commercial off-the-shelf (COTS) software and hardware.

Communication protocols are an integral element of space systems enabling command uplink, data relays, and platform and payload telemetry downlink from spacecraft (e.g., satellites, rovers). As for terrestrial systems, such protocols are also a potential vector for cyberattacks [12], [34]. Experience shows that implementations of communication protocols are often affected by coding defects, especially if those implementations are written in memory-unsafe languages like C and C++, which are widely used in space systems. Furthermore, space systems typically use dedicated protocols that have received relatively little scrutiny from the security community [44]. For example, protocols standardized by the Consultative Committee for Space Data Systems (CCSDS) [7] such as the Bundle Protocol (BP) [39], [6] are critical for initiatives that aim to build communication networks beyond Earth like Moonlight [11], LunaNet [38], and the Solar System Internet [1].

Fuzzing is a popular software testing technique for discovering vulnerabilities by feeding large amounts of inputs to a target program [27]. While there exists a wealth of research on fuzzing [25], [47], most research focuses on fuzzing software that takes files as inputs, with the research on fuzzing network protocols being comparatively smaller [36], [3], [30], [4]. Few works have explored the fuzzing of space systems [17], [37]. Gutierrez et al. [17] developed a custom fuzzer for the SUCHAI CubeSat platform, Scharnowski et al. [37] fuzzed three satellite firmware images, and Willbold et al. [43] discussed the challenges fuzzing satellite firmware. Those works fuzzed the satellites' telecommand interfaces, but many other space (communication) protocols exist.

This work reports our preliminary experiences when fuzzing five open-source implementations of four space protocols using two fuzzing approaches: grammar-based fuzzing with Peach [19] and coverage-guided fuzzing with AFL++ [13]. To enable the fuzzing, we developed grammars for the protocols and custom harnesses for the targets. Our fuzzing campaigns identified 11 vulnerabilities in four targets. The vulnerabilities have been disclosed responsibly to the affected projects. So far, 5 vulnerabilities have been patched and 4 have been assigned CVE identifiers. We also found a discrepancy in how one target interpreted the protocol standard, which we reported to the project and has since been fixed.

II. Selection Process

This section describes our selection process for (1) space protocols to be tested, (2) open-source implementations of those space protocols to use as fuzzing targets, and (3) fuzzing techniques to apply for finding vulnerabilities in those targets.

A. Protocol Selection

The main protocol selection criteria were that the protocols were used by currently flying satellites and that they operate at the network layer and above, avoiding link layer protocols that may require radio interfaces. We focus on three network layer standardized protocols from CCSDS [7], which are utilized by member space agencies (e.g., NASA, ESA). We also include a protocol that, while not standardized, has an official implementation that has been operated in multiple missions. Figure 1 shows the stacking of the selected protocols.

BPv6 and BPv7. The Bundle Protocol (BP) is a key protocol of the Delay/Disruption-Tolerant Networking (DTN) architecture, designed for reliable data transmission across networks

Workshop on Security of Space and Satellite Systems (SpaceSec) 2025
24 February 2025, San Diego, CA, USA
ISBN 979-8-9919276-1-1
https://dx.doi.org/10.14722/spacesec.2025.23012
www.ndss-symposium.org

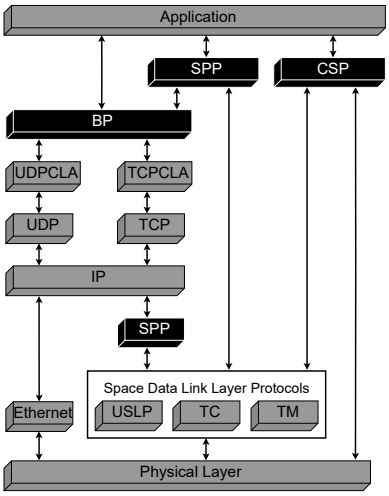

Figure 1: Protocol stack with the selected protocols (in black).

with intermittent connectivity and long delays. By bundling data into packets that can be stored and forwarded through intermediate nodes, BP ensures information can eventually reach its destination even if direct end-to-end communication is temporarily unavailable. Two versions of BP (BPv6 and BPv7) are documented in RFCs [39], [6] and are included in the CCSDS protocol suite.

SPP. The Space Packet Protocol (SPP) is a CCSDS protocol that transfers data between spacecraft and ground system processes. It wraps telecommands and telemetry Protocol Data Units transferred by lower protocols. SPP is a much simpler protocol than BP. Two of the three satellites analyzed by Willbold et al. use SPP [44].

CSP. The CubeSat Space Protocol (CSP) is a lightweight network protocol designed for small satellites, particularly CubeSats [21]. CSP was developed in collaboration between Aalborg University and GomSpace and was first deployed in the GOMX-1 CubeSat in 2012 [2]. In November 2018, Klofas identified 23 CubeSats using CSP [20]. The SUCHAI CubeSat analyzed by Gutierrez et al. [17] and one of the three satellites analyzed by Willbold et al. use CSP [44].

Two challenges for fuzzing the selected protocols are the encodings used in both BP versions and the checksums used in BPv7 and SPP. In BPv6, integer fields are encoded using Self-Delimiting Numeric Values (SDNV) [10], while BPv7 encodes all fields using the Concise Binary Object Representation (CBOR) [5]. Due to the per-field encoding, it is not possible to separate decoding from parsing or encoding from building new inputs, as those are intertwined. Regarding checksums, BPv7 optionally uses X-25 CRC-16 or CRC32C (Castagnoli), and SPP uses a CCSDS-specified 16-bit CRC. We discuss how we address these challenges in Section III-B.

B. Target Selection

We searched for open-source implementations of the selected protocols, prioritizing those developed by space agencies or private organizations, those deployed in space, and those that are maintained (i.e., at least one update in the last year). We selected the five targets shown in Table I. They include two BP implementations from NASA (ION-DTN [29] and HDTN [28]), both tested in the International Space Station (ISS) [9]. We also include µD3TN, developed by the D3TN company, which supports both BP and SPP and is space-tested [8], [42]. For CSP, we select the official *libcsp* implementation, deployed in numerous CubeSat missions [21]. Finally, we include python-spp, which is written in Python and thus not designed to run on spacecraft but allows us to test non-C/C++ implementations.

Target	Org.	Lang.	SLOC	BPv6	BPv7	SPP	CSP
ION-DTN [29]	NASA	C	814K	✓	✓		
HDTN [28]	NASA	C++	81K	✓	✓		
µD3TN [15]	D3TN	C	51K	✓	✓	✓	
libcsp [21]	OSS	C	9K				✓
python-spp [22]	LibreCube	Python	275			✓	

Table I: Targets used in evaluation.

For all implementations, we set up a node that receives packets, typically by encapsulating the target protocol over UDP (TCP for python-spp). µD3TN provides stand-alone executables to run the protocol parsers on files, which simplifies the setup. ION-DTN is the most complex target, with 814K Source Lines Of Code (SLOC). Furthermore, it is multi-process, spawning eight processes on startup.

C. Fuzzer Selection

Grammar-based fuzzing [16] leverages protocol specifications for fuzzing. Two popular grammar-based network fuzzers are Peach [19] and BooFuzz [35]. These fuzzers do not have grammars for space protocols. To evaluate grammar-based fuzzing, we developed Peach grammars (called *pits*) for the four selected protocols, as described in Section III-A. We use the GitLab Protocol Fuzzer CE based on Peach Pro [19]. We also tried developing protocol grammars for BooFuzz, but this fuzzer does not handle fields shorter than 8 bits, which are common in the selected protocols [23].

Other network fuzzers focus on making coverage-guided fuzzing [46] state-aware by prioritizing inputs that discover new protocol states (e.g., AFLNet [36], SGFuzz [3], StateAFL [30]). We did not evaluate them since the selected protocols do not allow long message interactions. Bleem [24] uses a man-in-the-middle (MITM) approach that modifies the traffic in-flight between client and server, but the fuzzer is not publicly available. Recently, Fuzztruction-Net [4] proposed introducing faults during the execution of a peer, letting the peer handle the generation of inputs after the fault. This approach is the only one that could potentially handle the encoding in BP, but it was published at the end of our work.

We also evaluate AFL++[13], arguably the leading coverage-guided fuzzer. Since AFL++ does not support network protocol fuzzing, we only apply it for fuzzing two targets: the µD3TN executables that read inputs from files and HDTN through a custom harness that de-socketizes the code. AFL++ is significantly faster at producing inputs than Peach, but in the presence of encoding and checksums it may generate many invalid inputs that are quickly dropped by the parsers. Our Peach pits address those issues, but Peach's slowness compared

to AFL++ means those improvements may not compensate for the speed difference. We use AFL++ version 4.21a.

III. Fuzzing Setup

This section describes how we set up the fuzzing by developing protocol grammars and custom harnesses.

A. Peach Pits

To enable fuzzing with Peach we had to create pits for each protocol. 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 relations to other fields, such as length fields (used in all four protocols) and offset fields that capture the start offset of other fields (used in BPv6). To generate valid inputs, we developed C# Peach plugins that encode the fields and compute checksums. In particular, we developed *Transformers* for encoding fields with SDNV (BPv6) and CBOR (BPv7) and *Fixups* for computing the SPP and BPv7 checksums. Peach fixups are applied before transformers, but the BPv7 checksum is calculated over the CBOR-encoded fields. To address this issue, we added the checksum computation into the CBOR transformer.

For all targets except *libcsp*, the state model simply requires the fuzzer to send the built input to the target on top of either UDP or TCP. Instead, libcsp employs the ZeroMQ (ZMQ) publish-subscribe protocol. Thus, the state model requires the fuzzer to complete the ZMQ handshake prior to sending the CSP message. We added the ZMQ handshake messages to the data model, marking them as immutable so that only the CSP message would be fuzzed.

In theory, only one pit is needed for each protocol. In practice, we had to keep multiple similar pits for some protocols due to target differences and protocol features. For example, we found that µD3TN computed the value for the BPv6 Primary Block Length field differently than ION-DTN and HDTN. The issue is caused by a potentially confusing description in RFC 5050: "The Block Length field is an SDNV that contains the aggregate length of all remaining fields of the block." [39]. Both ION-DTN and HDTN interpreted this as excluding the fields preceding Block Length and the Block Length field itself, while µD3TN included those three fields in the length value. We reported this issue to µD3TN and it has been patched to match the interpretation of ION-DTN and HDTN [18]. We also created a separate BPv6 pit because if the BPv6 fragmentation flag is set, two additional fields are added to the BPv6 bundle structure.

B. Target Setup

We used Peach to fuzz all targets over the network. We used AFL++ to fuzz µD3TN and HDTN using input files with network message payloads. µD3TN provides stand-alone executables for its parsers. For HDTN, we implemented a harness that calls the main parsing function with the payload read from a file. We did not de-socketize other targets as it was not easy to pinpoint a function to use as an entry point for parsing. Furthermore, ION-DTN is multi-process and cannot be fuzzed with the off-the-shelf AFL++.

When fuzzing with AFL++, we leveraged its afl-cc/afl-c++ compilers with default settings, relying on the built-in coverage tracking and crash and hang detection capabilities. For µD3TN, we used sample network inputs provided in the project's repository as seeds. For HDTN, we set up two nodes, captured the exchanged messages, and used them as seeds.

When fuzzing with Peach, we did not use Peach's oracles because we could not compile our targets to run on Ubuntu 16.04 (the latest OS supported by Peach), thus having to run the targets on a separate machine. Instead, we created custom harnesses that fork the target and detect if it terminates. Since python-spp is written in Python, its harness also checks for uncaught exceptions. The harnesses also save the application logs, run tcpdump to collect network traces, and use Valgrind [31] to identify bugs that may not cause a crash. We tried using Address Sanitizer (ASan) [40] instead of Valgrind but could not get ION-DTN to compile with ASan. To get coverage data, we compiled the targets with gcov [14] support.

ION-DTN was especially challenging to set up, perhaps unsurprisingly, as its deployment guide states, "ION is generally optimized for continuous operational use rather than research... Unfortunately, this can make ION somewhat painful for new users to work with..." [29]. The main issues were that the ION core uses eight processes and, when terminated, it does not entirely release its resources. To address this issue, we had to customize its harness with custom reset scripts that released resources that prevented ION-DTN from restarting.

IV. Evaluation

Table II summarizes our fuzzing campaigns. We fuzzed the latest version of each target at the time. When patches for vulnerabilities we discovered were released during our experiments, we also fuzzed the patched version. We tried running 24-hour fuzzing campaigns. However, we had stability issues with the Peach harnesses for ION-DTN and HDTN, which caused their campaigns to terminate earlier. In the case of ION-DTN this is likely due to memory leaks caused by an incomplete reset process. Peach generated 77K–85K inputs per hour and AFL++ 12.7M–35M inputs per hour, two orders of magnitude more. Our campaigns found from zero (libcsp) up to 3M (µD3TN) crashes. We tested crashing inputs on a full target node with Valgrind to validate the crashes and generate stack traces. We bucketed similar stack traces into 11 unique vulnerabilities, summarized in Table III.

The 11 vulnerabilities belong to common categories of software security issues such as out-of-bounds reads, double-frees, and use of uninitialized variables. The vulnerability in python-spp was notably different. Malformed packets caused an exception, and the thread responsible for receiving packets would terminate silently while the main server continued running, unaware that it was no longer receiving data. All vulnerabilities scored 7.5 on the CVSS v3 rating. These high scores are due to the network attack vector, the low attack complexity, and the absence of any requirements for privileges or user interaction. Each vulnerability, at a minimum, results in a denial-of-service (DoS) for the affected network service.

All vulnerabilities have been responsibly disclosed. For µD3TN, we directly reported the issues through the GitLab private reporting functionality. For the other targets, we

Target	Version	Protocol	Fuzzer	Inputs	Time	GCOV Coverage	AFL++ Coverage	Crashes	Vulnerabilities
ION-DTN [29]	4.1.3	BPv6	Peach	155k	2 hrs	13.9%	-	10	3
	4.1.3	BPv7	Peach	510k	6 hrs	18.3%	-	123	2
HDTN [28]	1.3.1	BPv6	Peach	1.1M	13 hrs	1.9%	-	-	-
	1.3.1	BPv6	AFL++	21.1M	24 hrs	-	1.8%	4,215	3
	1.3.1	BPv7	Peach	642k	10 hrs	2.2%	-	-	-
µD3TN [15]	0.13.0	SPP	AFL++	306M	24 hrs	-	1.2%	-	-
	0.13.0	SPP	Peach	1.7M	24 hrs	1.2%	-	-	-
	0.13.0	BPv6	AFL++	693M	24 hrs	-	4.9%	-	-
	0.13.0	BPv6	Peach	1.8M	24 hrs	4.6%	-	-	-
	0.13.0	BPv7	AFL++	648M	24 hrs	-	7.9%	387	1
	0.13.0	BPv7	Peach	1.7M	24 hrs	6.0%	-	-	-
	0.14.1	BPv7	AFL++	849M	24 hrs	-	6.6%	3.0M	1
	0.14.2	BPv7	AFL++	843M	24 hrs	-	6.9%	-	-
libcsp [21]	2.0	CSP	Peach	1.8M	24 hrs	30.2%	-	-	-
python-spp [22]	5cbffe	SPP	Peach	401K	24 hrs	-	-	1	1

Table II: Fuzzing campaigns conducted for each target.

Target	Ver.	Prot.	Status	Vulnerability Type
ION-DTN	4.1.3	BPv7	CVE-2024-54130	CWE-457: Use Uninitialized Variable
ION-DTN	4.1.3	BPv7	CVE-2024-54129	CWE-665: Improper Initialization
ION-DTN	4.1.3	BPv6	reported	CWE-125: Out-of-bounds Read
ION-DTN	4.1.3	BPv6	reported	CWE-20: Improper Input Validation
ION-DTN	4.1.3	BPv6	reported	CWE-125: Out-of-bounds Read
HDTN	1.3.1	BPv6	reported	CWE-122: Heap Buffer Overflow
HDTN	1.3.1	BPv6	reported	CWE-457: Use Uninitialized Variable
HDTN	1.3.1	BPv6	reported	CWE-789: Uncontrolled Mem. Alloc.
µD3TN	0.13.0	BPv7	CVE-2024-10455	CWE-617: Reachable Assertion
µD3TN	0.14.1	BPv7	CVE-2024-12107	CWE-415: Double Free
LibreCube	5cbffe	SPP	fixed	CWE-248: Uncaught Exception

Table III: Vulnerabilities found and reporting status.

approached the developers privately asking them how they wanted the issue reported. For ION-DTN, we guided the responsible NASA team on enabling the private reporting options in GitHub. Five vulnerabilities have already been patched and 4 vulnerabilities have been assigned CVE identifiers.

We briefly describe two vulnerabilities that have already been patched by the developers.

Double-Free in µD3TN BPv7. A double-free vulnerability could be triggered in µD3TN through malformed BPv7 Endpoint Identifiers (EIDs). In our setup, the *glibc tcache* memory protection detected the double-free and aborted the process, resulting in a DoS. However, given resource constraints, spacecraft may use lightweight C standard libraries like *uClibc* [41] or *newlib* [32], which may lack such memory protections. In such environments, a double-free might go undetected, potentially leading to undefined behavior and further exploitation.

Use of Uninitialized Variables in ION-DTN. Setting the Destination Endpoint Identifier (EID) of a BPv7 bundle to "dtn:none" caused ION-DTN to crash. Essentially, the EID was treated as valid, but the subsequent logic did not handle the special "none" case correctly, leading to some variables being left uninitialized. This resulted in a segmentation fault when those uninitialized variables were later accessed.

V. Discussion

Fuzzer comparison. Both fuzzers found similar numbers of vulnerabilities: 6 for Peach in two targets (ION-DTN, python-spp) and 5 for AFL++ in two targets (HDTN, µD3TN). However, AFL++ showed its superiority by generating two orders of magnitude more inputs per hour and finding vulnerabilities that Peach did not find, which agrees with prior results [36], [3], [30]. However, fuzzing network protocols with AFL++ required de-socketizing the targets and was problematic with the multi-process ION-DTN, with LibreCube's Python implementation, and for libcsp that required an initial message handshake. Thus, we found value in using both fuzzers. Furthermore, Peach's limitations may not be intrinsic to its grammar-based approach but to its lack of coverage-guided support and its outdated implementation. Furthermore, the developed protocol grammars could enable other applications such as intrusion detection and anomaly detection.

Source code availability. We have fuzzed open-source targets. Source code is typically available to the developers, but it may not be available to external analysts. Its availability allows more straightforward modification of a target, e.g., to create the harness that de-socketizes HDTN. Creating such stand-alone executables is harder without source code. For example, Willbold et al. [43] failed to create a stand-alone executable of the telecommand parsing code from the firmware of the Flying Laptop satellite.

Fuzzing setup. Most of our time was spent setting up the targets for fuzzing, which was difficult due to the steep learning curve and the limited documentation of some targets. We observe that developers can greatly assist in the process of security testing their software by providing stand-alone executables for the protocol parsers, better documentation, and making sure that all used resources are released upon termination. Facilitating the security testing process results in more resilient implementations.

VI. Conclusion

We have reported our experiences when fuzzing open-source space protocol implementations using two popular fuzzing approaches. We identified 11 remotely exploitable vulnerabilities in four out of five targets. Our results highlight that security testing of space systems should be a priority. A clear avenue to improve this work would be expanding its coverage by examining more space protocols (e.g., link layer, application layer), fuzzing more targets (e.g., closed-source implementations), and evaluating more fuzzers (e.g., AFLNet [36], Fuzztruction-Net [4]).

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

[2] L. Alminde, J. Christiansen, K. Kaas Laursen, A. Midtgaard, M. Bisgard, M. Jensen, B. Gosvig, A. Birklykke, P. Koch, and Y. Le Moullec, "Gomx-1: A nano-satellite mission to demonstrate improved situational awareness for air traffic control," in *Annual AIAA/USU Conference on Small Satellites*, 2012.

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

[4] 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.

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

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

[7] "Consultative committee for space data systems," 2024. [Online]. Available: https://public.ccsds.org/default.aspx

[8] D3TN, "BPv7 field-tests (IETF 110 Meeting / DTN WG / 2021-03-11)," November 2021. [Online]. Available: https://datatracker.ietf.org/meeting/110/materials/slides-110-dtn-bpv7-field-tests-00

[9] R. Dudukovich, D. Raible, B. Tomko, N. Kortas, E. Schweinsberg, T. Basciano, W. Pohlchuck, J. Deaton, J. Nowakowski, and A. Hylton, "Advances in high-rate delay tolerant networking on-board the international space station," in *IEEE Space Computing Conference*, 2024.

[10] W. Eddy and E. B. Davies, "Using Self-Delimiting Numeric Values in Protocols," RFC 6256, May 2011.

[11] 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

[12] G. Falco, "The vacuum of space cyber security," in *AIAA SPACE and Astronautics Forum and Exposition*, 2018.

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

[14] "gcov—a test coverage program," 2024. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[15] D. GmbH, "ud3tn," 2024, accessed: November, 2024. [Online]. Available: https://gitlab.com/d3tn/ud3tn

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

[17] T. Gutierrez, A. Bergel, C. E. Gonzalez, C. J. Rojas, and M. A. Diaz, "Systematic fuzz testing techniques on a nanosatellite flight software for agile mission development," *IEEE Access*, vol. 9, 2021.

[18] S. Havermans, "Deviation in BPv6 Primary Block Length Field in uD3TN," December 2024. [Online]. Available: https://gitlab.com/d3tn/ud3tn/-/issues/236

[19] G. Inc., "Gitlab protocol fuzzer community edition," https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce, 2024, accessed: November, 2024.

[20] B. Klofas, "Cubesat communications system table," November 2018. [Online]. Available: https://www.klofas.com/comm-table/

[21] libcsp contributors, "libcsp," 2024, accessed: November, 2024. [Online]. Available: https://github.com/libcsp/libcsp

[22] LibreCube, "Python spacepacket," 2024, accessed: November, 2024. [Online]. Available: https://gitlab.com/librecube/lib/python-spacepacket/-/tree/5cbffe77a963ade0215d983877f75a24cd0caa69

[23] lockout, "Fuzz a bitfield of various fixed bit sizes," September 2016. [Online]. Available: https://github.com/jtpereyda/boofuzz/issues/88

[24] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet sequence oriented fuzzing for protocol implementations," in *USENIX Security Symposium*, 2023.

[25] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, 2019.

[26] M. Manulis, C. P. Bridges, R. Harrison, V. Sekar, and A. Davis, "Cyber security in new space: analysis of threats, key enabling technologies and challenges," *International Journal of Information Security*, vol. 20, pp. 287–311, 2021.

[27] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of unix utilities and services," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1995.

[28] NASA, "High data rate tolerant network (hdtn) software," 2024, accessed: November, 2024. [Online]. Available: https://github.com/nasa/HDTN/tree/main

[29] NASA-JPL, "Ion-dtn," 2024, accessed: November, 2024. [Online]. Available: https://github.com/nasa-jpl/ION-DTN

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

[31] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[32] "newlib," 2024. [Online]. Available: https://sourceware.org/newlib/

[33] J. Pavur and I. Martinovic, "Building a launchpad for satellite cybersecurity research: lessons from 60 years of spaceflight," *Journal of Cybersecurity*, vol. 8, no. 1, p. tyac008, 2022.

[34] J. Pavur, D. Moser, V. Lenders, and I. Martinovic, "Secrets in the sky: on privacy and infrastructure security in dvb-s satellite broadband," in *Security and Privacy in Wireless and Mobile Networks*, 2019.

[35] J. Pereyda, "Boofuzz," https://github.com/jtpereyda/boofuzz, 2024, accessed: November, 2024.

[36] 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.

[37] T. Scharnowski, F. Buchmann, S. Wörner, and T. Holz, "A case study on fuzzing satellite firmware," in *Workshop on the Security of Space and Satellite Systems*, 2023.

[38] 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/

[39] K. Scott and S. C. Burleigh, "Bundle Protocol Specification," RFC 5050, Nov. 2007.

[40] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *USENIX Annual Technical Conference*, 2012.

[41] "uclibc," 2024. [Online]. Available: https://uclibc.org/

[42] F. Walter, M. Feldmann, J. A. Fraire, and S. Burleigh, "The architectural refinement of µD3TN: Toward a software-defined dtn protocol stack," *arXiv preprint arXiv:2407.17166*, 2024.

[43] J. Willbold, M. Schloegel, F. Göhler, T. Scharnowski, N. Bars, S. Wörner, N. Schiller, and T. Holz, "Scaling software security analysis to satellites: Automated fuzz testing and its unique challenges," in *IEEE Aerospace Conference*, 2024.

[44] J. Willbold, M. Schloegel, M. Vögele, M. Gerhardt, T. Holz, and A. Abbasi, "Space odyssey: An experimental software security analysis of satellites," in *IEEE Symposium on Security and Privacy*, 2023.

[45] N. Yadav, F. Vollmer, A.-R. Sadeghi, G. Smaragdakis, and A. Voulimeneas, "Orbital shield: Rethinking satellite security in the commercial off-the-shelf era," in *Security for Space Systems (3S)*, 2024.

[46] M. Zalewski, "American fuzzy lop (afl)," http://lcamtuf.coredump.cx/afl, 2017, accessed: November, 2024.

[47] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–36, 2022.