NFDFuzz: A Stateful Structure-Aware Fuzzer for Named Data Networking

George Torres
National Institute of Standards and Technology
Gaithersburg, MD, USA
george.torres@nist.gov

Davide Pesavento
National Institute of Standards and Technology
Gaithersburg, MD, USA
davide.pesavento@nist.gov

Junxiao Shi
National Institute of Standards and Technology
Gaithersburg, MD, USA
junxiao.shi@nist.gov

Lotfi Benmohamed
National Institute of Standards and Technology
Gaithersburg, MD, USA
lotfi.benmohamed@nist.gov

ABSTRACT

Fuzzing is a very popular automated testing technique that has yet to be applied in any significant way to NDN (Named Data Networking). NDN and its software forwarding daemon NFD present interesting challenges for fuzzing. To be effective, a fuzzer for NFD needs to be both stateful, due to the nature of the NDN data plane, and aware of the packet structure and the rules governing the NDN wire protocol. In this work we present the design of our NFD fuzzer and provide an overview of its most salient implementation details.

CCS CONCEPTS

• **Networks** → **Protocol testing and verification**; *Routers*; Network layer protocols; • **Software and its engineering** → **Software testing and debugging**; *Software reliability*.

KEYWORDS

Named data networking, NDN forwarder, Software testing, Automated fuzz testing, Stateful fuzzing, Structure-aware fuzzing

ACM Reference Format:
George Torres, Davide Pesavento, Junxiao Shi, and Lotfi Benmohamed. 2020. NFDFuzz: A Stateful Structure-Aware Fuzzer for Named Data Networking. In *7th ACM Conference on Information-Centric Networking (ICN '20), September 29–October 1, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3405656.3420234

1 INTRODUCTION

Fuzz testing (or *fuzzing*) is an automated software testing technique in which invalid and/or unexpected inputs are randomly generated and fed to a program. This process is managed by a software tool known as *fuzzer* which will repeat this process until it identifies a potential bug in the program under test. Fuzzing is a useful technique for finding software defects that are overlooked when that software was initially written. For example, Google has used fuzz testing to find thousands of security vulnerabilities [6].

ICN '20, September 29–October 1, 2020, Virtual Event, Canada
2020. ACM ISBN 978-1-4503-8040-9/20/09.
https://doi.org/10.1145/3405656.3420234

There are two main types of fuzzers: *generation-based* fuzzers and *coverage-guided mutation-based* fuzzers [2]. The former generate random inputs according to a predefined grammar, while the latter start from existing inputs (*corpus*) and mutate them to create new inputs for the program under test. Any inputs that trigger new execution paths that were previously uncovered are saved back into the corpus. A mutation-based fuzzer can be *dumb* or *smart*. A dumb fuzzer will blindly mutate an input, with no regard to its overall structure. These fuzzers typically only stress the parsing routines of programs that have complicated or rigidly structured input types because a dumb mutation usually leads to an invalid input that is rejected in the early stages of parsing. A smart fuzzer, also known as *structure-aware*, takes the input's structure into consideration when making mutations. This enables the fuzzer to create inputs that can pass the parsing stage and trigger more code paths, thereby substantially increasing the testing coverage.

Incorporating fuzz testing into a project such as Named Data Networking (NDN) can greatly improve the ability to uncover any hidden exploitable bugs. NDN [1, 7] is a proposed future Internet architecture in which IP addresses are replaced by hierarchical names. These names are used by the NDN Forwarding Daemon (NFD) [5] to forward packets to the appropriate destination. NFD is a critical component of NDN and the part we wish to fuzz test.

There are several challenges that must be addressed in order to efficiently fuzz an NDN forwarder such as NFD. Firstly, the forwarder's inputs (Interest and Data packets) are highly structured, thus requiring a structure-aware fuzzer capable of producing packets that are "valid enough" to be accepted by the parser and therefore able to stress the later stages of the forwarding logic. Secondly, NDN has a *stateful* data plane. As a consequence, the fuzzer must also be stateful and maintain an internal history of packets that were recently sent to the forwarder, enabling us to test unique NDN features such as Interest aggregation, multicasting, and Data caching. Thirdly, a single packet is rarely sufficient to trigger a bug in the forwarder's code; more typically multiple packets in a specific order are necessary to force NFD into an erroneous state where it misbehaves or crashes. Therefore, the traditional concept of "input" must be generalized to encompass a *sequence* of packets.

In summary, to effectively fuzz an NDN forwarder, we need a coverage-guided fuzzer that is both structure-aware and stateful. These are the main design principles on which **NFDFuzz**, our NFD fuzzer, is based. In the remainder of this paper we describe the

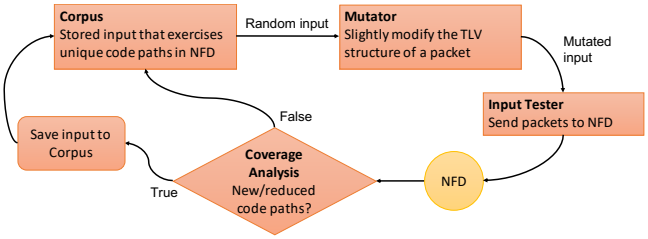

Figure 1: Main fuzzing cycle.

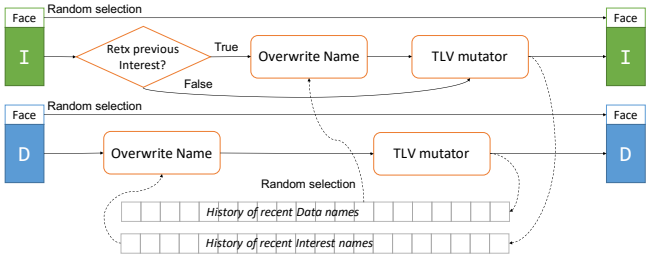

Figure 2: High-level mutator.

most important aspects of its architecture and implementation. Preliminary results are very promising: an early-stage prototype was able to find two bugs in NFD in just a few hours.

2 NFDFUZZ DESIGN OVERVIEW

NFDFuzz can be described as having three distinct phases: the initialization, setup, and main phases. One of the most important requirements of NFDFuzz is the initialization and maintenance of NFD throughout the testing process. This is done in the initialization phase, after which NFD will continue to run in the background. The setup phase is where all the necessary data structures are created and NFD is configured. First, an instance of each type of face is created. Then, a set number of prefixes for Data names are randomly generated and stored for later use. The FIB is populated with these prefixes, each will have a random face as next hop. The final step is to establish the use of all available forwarding strategies. To accomplish this, each prefix is assigned to one of the strategies.

After the setup we will move on to the main phase (fig. 1) which runs in an endless loop until a bug is detected. The two most important components of this phase are the *mutator* and the *input tester*. The mutator is designed to generate new inputs (packets) for NFD and feed them into the tester, which will then send them to NFD on the appropriate face(s). These inputs are also stored into input traces that are created as the fuzzer runs, so that the entire sequence of events leading to a particular program state can be replayed, allowing for accurate coverage gathering and for aiding the process of identifying possible causes for a crash. Individual inputs are also saved if they were considered "interesting", meaning they found new execution paths in the code.

The mutator can be broken down into a *high-level* mutator and a *low-level* mutator. The low-level mutator takes an Interest or Data packet and makes a random change to one of its TLV elements. The exact kind of mutation applied to the packet is randomly selected among the following ones: add a TLV element, remove a TLV element, change the TLV type of an element, shuffle the elements' order, mutate a (randomly chosen) sub-element. In the last case, if the chosen TLV element is simple, the element's bytes will be passed to a "dumb" fuzzer to undergo a basic heuristic mutation. Otherwise, if the chosen element has its own TLV-based structure, the low-level mutator will recursively apply itself to that element.

The high-level mutator (fig. 2) is vaguely similar to a traffic generator: it understands the NDN protocol semantics and will make decisions on how and what packets will be emitted. The mutator will randomly choose a face for sending and the prefix that will be attached to the generated Interest packet. Based on the state of NFD, it can also choose to send a Data reply or retransmit a previously sent Interest. This mutator will also direct the low-level mutator to alter a packet.

3 IMPLEMENTATION DETAILS

We are using a modified `libFuzzer` libray [3] in our implementation of NFDFuzz. The library is modified so that the setup phase can be correctly initialized. From there we defined our custom high-level and low-level mutators. The high-level mutator is implemented inside of the basic template function provided by `libFuzzer` to create a custom mutator. The low-level mutator is implemented via a collection of unique functions, designed to produce a TLV structure that is able to pass through the NDN packet parser.

The fuzzer requires multiple thread to work together in order to run correctly. The fuzzer first splits into two threads: the main thread that will drive the main phase and a background thread that will handle the initialization phase, setup phase, and the maintenance of NFD. This background thread will be again split into two threads: an NFD thread that starts up and maintains NFD in the background, and a setup thread that will complete the creation of the NFD structures which will be under test. The setup thread will wait for NFD to be up and running before creating the data structures. The faces that are created will have a reference stored in a map for use by the input during the testing phase. The name prefixes are stored inside a vector for use in the mutator. Once the setup work is done, this thread will merge with the NFD thread.

4 FUTURE WORK

We would like to improve upon the fuzzer by adding more complexity and scope via additional modules, such as the NFD management protocol. Other future additions include support for various NDN Link Protocol (NDNLP) [4] features such as fragmentation/reassembly, reliability, and Nacks. In the longer term, we envision a hybrid approach between mutation-based and generation-based fuzzers so that valid NDN packets can be automatically generated from a machine-readable description of the protocol.

DISCLAIMER

Any mention of commercial products or reference to commercial organizations is for information only; it does not imply recommendation or endorsement by NIST, nor does it imply that the products mentioned are necessarily the best available for the purpose.

REFERENCES

[1] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. 2009. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. 1–12.

[2] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 6.

[3] LLVM Project. [n.d.]. *A library for coverage-guided fuzz testing.* Retrieved August 31, 2020 from https://llvm.org/docs/LibFuzzer.html

[4] Named Data Networking Project. [n.d.]. *NDNLPv2: NDN Link Protocol, version 2.* Retrieved August 31, 2020 from https://redmine.named-data.net/projects/nfd/wiki/NDNLPv2

[5] Named Data Networking Project. 2018. *NFD Developer's Guide.* Technical Report. NDN-0021, Revision 10. https://named-data.net/publications/techreports/ndn-0021-10-nfd-developer-guide/

[6] Matt Ruhstaller and Oliver Chang. [n.d.]. *A New Chapter for OSS-Fuzz.* Retrieved August 31, 2020 from https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html

[7] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, KC Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 66–73.