NESFuzzer: Stateful Fuzz Testing of IoT Network Stacks

Nicolas Tsiftes
RISE Research Institutes of Sweden and Digital Futures
Stockholm, Sweden
nicolas.tsiftes@ri.se

Thiemo Voigt
RISE Research Institutes of Sweden
Stockholm, Sweden
Uppsala University
Uppsala, Sweden
thiemo.voigt@angstrom.uu.se

Abstract

Low-power IPv6 stacks are being used in a myriad of IoT networks, which makes it essential that they are secure. Fuzz testing has proven an invaluable method to discover vulnerabilities in software that take complex input, including protocol implementations. Still, fuzzing tools typically cannot test state-dependent parts of IoT network stacks. To bridge this gap, we design and implement NESFuzzer, a tool for stateful fuzzing of IoT network stacks. Unlike previous methods in this area, NESFuzzer enables stateful fuzzing while leveraging the techniques of existing coverage-guided and hybrid fuzzing tools such as MOpt and SymCC without modifications. This decoupling is made possible through a modular, network-based architecture in which the system being tested communicates with an external service to generate specific protocol states. We evaluate our work through a case study on the Contiki-NG operating system, and show that the test coverage increases by up to 170% in certain modules of the network stack. Using NESFuzzer, we have discovered four security vulnerabilities that could not be found with conventional fuzzing.

CCS Concepts

• **Security and privacy → Operating systems security**; • **Computer systems organization → Embedded software**; • **Software and its engineering → Software testing and debugging**.

Keywords

Internet of Things, fuzz testing, embedded network stacks, stateful

1 Introduction

Numerous security vulnerabilities in IoT devices have been exposed in recent years. For instance, a report released in 2020 shows vulnerabilities in several network stacks for low-power wireless communication [12]. The network stack of a resource-constrained IoT operating system consists of various protocols typically organized in layers, with IPv6 being the *narrow waist* [17]. Hence, for a single input packet, different protocols may process different parts of the packet. Vulnerabilities in protocol implementations can be exploited by attackers if they are able to transmit carefully crafted packets into the network stack [4, 16].

Fuzz testing, or *fuzzing*, is an effective dynamic method to expose vulnerabilities and bugs in software through the injection of a huge amount of mutated input data [22, 35]. It is especially useful for testing software that processes complex input data—a key trait of an IoT network stack. Figure 1 shows a common fuzzing setup, where a fuzzing tool, or *fuzzer*, injects a large number of mutated data into a System Under Test (SUT). Moreover, a *fuzzing harness* may be

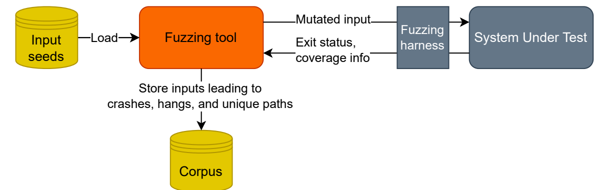

Figure 1: A common fuzzing setup consisting of a fuzzing tool that executes a System Under Test iteratively with a huge amount of mutated input data. The main objective is to reach many different execution paths and expose bugs.

employed with the SUT to direct the input to a specific entry point in the code and steer its execution to suit the test objective. This type of fuzzing is typically stateless, as a single input is given for each execution of the SUT. In SUTs that have state-dependent code, such as an IPv6-based IoT network stack, a regular fuzzing setup limits the attainable test coverage [31]. Hence, one cannot test code segments that can be reached only when the SUT is in some specific state. For example, a single TCP connection can transition among a variety of states [37]. Therefore, one needs to be able to establish such states in a controlled manner to achieve a high coverage when fuzzing the TCP implementation of an IoT network stack.

Due to the lack of practical tools to generate state for multiple protocols in IoT network stacks, previous fuzz testing campaigns have primarily been conducted using stateless fuzzing methods with state-of-the-art fuzzers. When fuzzing a network stack, it can be sufficient to inject a single packet per fuzzing iteration to test a large set of the code paths in the network stack that are not state-dependent. For instance, Poncelet et al. found 18 vulnerabilities in the network stack of the Contiki-NG OS using stateless fuzzing [31]. Hence, the next natural step to further improve the fuzzing of IoT network stacks is to enable stateful fuzzing.

Challenges and State of the Art. A variety of different tools have been proposed to overcome the limitations of current fuzzing methods when applied on state-dependent network protocol implementations [24, 29, 32]. Such tools are typically designed for testing applications that communicate using a single protocol over network sockets. By contrast, fuzzing an IoT network stack poses new challenges that are not addressed in existing tools for stateful fuzzing. In particular, a) the mutated input provided by fuzzers is processed by multiple protocol implementations operating at different layers, b) the protocols do not always offer a feedback mechanism that signals whether the internal state has changed (e.g., 6LoWPAN fragmentation), and c) the state in one protocol

implementation can sometimes affect the code paths taken in other protocol implementations.

Furthermore, existing stateful fuzzing approaches are typically strongly coupled with a specific fuzzing tool [29], and communicate with the SUT using protocol-controlled timers and asynchronous communication over network sockets [5]. Such communication can slow down the fuzzing considerably, counteracting one of the main advantages of fuzzing over other software testing methods such as symbolic execution: rapid software execution.

Approach. In this paper, we present NESFuzzer, a novel tool that enables stateful fuzz testing of embedded network stacks. A key aspect of NESFuzzer is that it separates the fuzz testing functionality from the state generation in a modular design, without losing the ability to employ the capabilities of state-of-the-art fuzzing tools. The state generation is handled by a separate process, called the *state controller*, that can exchange messages with the SUT to generate a set of configured states before injecting the fuzzing data. While this message exchange adds an overhead for the execution speed of the fuzzing, it trades it for an ability to reach new code paths in the state-dependent parts of protocol implementations. However, there are also optimizations that can reduce the SUT initialization overhead, as with the *deferred mode* available in a variety of AFL-based fuzzers [39]. Moreover, NESFuzzer is designed to be independent of the method to generate states to fuzz test. We equip the state controller with a set of protocol state models, which are implemented with the help of Scapy, an external library for parsing and generating messages for a large set of protocols [33].

When using NESFuzzer, the SUT needs to be modified slightly to connect the state controller with input and output functions that can be used to exchange messages to control the state in the SUT. This workflow is simplified through a software library provided as part of NESFuzzer, which has a set of function calls to handle the communication with the SUT and failure handling. When fuzzing the SUT, one can thus simply use a selected fuzzing tool and gain the state generation capability of NESFuzzer without having to modify the fuzzing tool.

Contributions. With this paper, we make the following technical contributions.

1. We present NESFuzzer, a novel method for fuzzing stateful protocols in IoT network stacks. NESFuzzer improves upon the state of the art through a programmable, decentralized design that decouples the state generation from the fuzzing tool, and supports fuzzing with configurable protocol states.
2. We experimentally evaluate NESFuzzer's efficacy to fuzz test an IoT network stack compared to conventional fuzzing, and show considerable improvements in the coverage of several protocol implementations. Our results show an increase of the coverage by up to 170% in certain modules of the Contiki-NG network stack compared to using the state-of-the-art fuzzing tool SymCC [30] alone.
3. We perform a case study on the Contiki-NG network stack, and find several new bugs in its RPL and TCP implementations, of which three are zero-day vulnerabilities that could not be found with state-of-the-art stateless fuzzers.

Outline. The remainder of the paper is structured as follows. We give background information on fuzzing and embedded network stacks in Section 2. Thereafter, we present our system design in Section 3, and our case study on the Contiki-NG OS in Section 4. We experimentally evaluate our work in Section 5, and cover related work in Section 6. Lastly, we conclude the paper in Section 7.

2 Background

This work is motivated by the need for increased security in embedded network stacks for the Internet of Things. In the following, we give an overview of such network stacks and how fuzz testing methods can be applied on them.

2.1 Fuzz Testing

Fuzz testing, or *fuzzing*, is a dynamic method for testing software implementations for execution errors such as crashes and timeouts [25]. A *fuzzing tool*, or *fuzzer* executes a *System Under Test* many times with different input data that has been generated by using a set of mutation operators. At the start of the test, the fuzzer can use a *test seed* consisting of input data files that are suitable for the SUT; e.g., a set of PDF document files when the SUT is a PDF processing library. It can also be possible to start without a test seed, leaving it to the fuzzer to try to generate suitable input data for the SUT. During a fuzzing session, the fuzzer typically stores a corpus of input data that generated an interesting response from the SUT, such as a crash, a hang, or a unique path found. This corpus can then be used for further analysis and testing of the SUT using debugging tools and code sanitizers.

In the simplest fuzzing method, called *blackbox fuzzing*, the fuzzing tool has no insight into the execution of the SUT. This can be the case when fuzzing a binary without having access to the source code. *Coverage-guided greybox fuzzing* is a method that can give considerable improvements in the fuzzing efficiency compared to blackbox fuzzing, as it can check which input mutations lead to new coverage following each execution [39]. It requires that the SUT is instrumented to allow the fuzzer to retrieve coverage data generated from each execution of the SUT. Such instrumentation can be implemented by adding instructions that write data in a coverage map when transitioning between basic blocks, or by executing the SUT in an emulator that can keep track of the coverage. The coverage map resides in shared memory that is mapped by both the fuzzing tool and the SUT. Fuzzing tools can also integrate other software testing methods such as symbolic execution [30] in what is called *hybrid fuzzing* [28]. Hybrid fuzzers can use these methods to generate new test cases when the mutation-based fuzzing cannot find new execution paths. The mutation-based fuzzer can then continue using this data for its subsequent mutations and thereby enable the testing to reach deeper into the SUT.

2.2 Embedded Network Stacks

Embedded network stacks are designed for resource-constrained devices whose typical requirements include low-power communication, low memory consumption, and small code size. Several such network stacks have been developed over the past two decades, including the Arch Rock IPv6 stack [17], the Berkeley Low-power IP stack (BLIP) [20], the Contiki-NG network stack [27], the Generic Network Stack (GNRC) in RIOT OS [6], lwIP [1], OpenThread [18], OpenWSN [38], and the Zephyr network stack [3].

These network stacks are typically based on multiple standard protocols at different communication layers. IPv6 commonly serves as the *narrow waist* at the network layer [17], providing a large 128-bit address space and auto-configuration for devices. 6LoWPAN serves as an adaptation layer below the IPv6 layer, where it handles header compression and packet fragmentation to support low-power and lossy networks (LLNs). For Neighbor Discovery (ND), one can use either the regular IPv6 ND or the LLN-optimized 6LoWPAN ND. For routing, the IETF has standardized IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [14]. RPL arranges the network nodes in a destination-oriented directed acyclic graph with distance-vector routing, and exchanges control messages using the Internet Control Message Protocol (ICMPv6).

For the transport layer, the network stacks typically provide UDP communication, but in a few cases also TCP. The Constrained Application Protocol (CoAP) is a standard protocol for the application layer that provides a request/response communication pattern similar to HTTP, but is designed for resource-constrained devices and datagram-oriented transport protocols such as UDP [36]. TCP is also included in several embedded network stacks, despite the challenges that resource constraints and the lossy links of low-power wireless communication can pose for TCP [7]. In fact, recent literature has shown how to implement and configure TCP in a way that ameliorates its performance in LLNs [21]. Several cryptographic protocols and services suitable for IoT networks are also standardized by the IETF, including Datagram Transport Layer Security (DTLS), Object Security for Constrained RESTful Environments (OSCORE), and Ephemeral Diffie-Hellman Over COSE (EDHOC). DTLS and OSCORE can be used with CoAP to provide secure end-to-end communication for applications, whereas EDHOC provides a lightweight protocol for Diffie-Hellman key exchange.

2.2.1 Fuzzing Support for Embedded Network Stacks. The plethora of protocols available in an embedded network stack pose some challenges for fuzzing. First, there can be dependencies between the protocols, making it difficult to fuzz test some protocol implementations in isolation. When fuzzing the network stack at some entry point for the fuzzing input, the network stack will make a series of validity checks on the input that is treated as a network packet. For example, it will first validate the 6LoWPAN header, then decompress the packet and feed it to the IPv6 implementation, where it will face another set of checks, and so on as the packet traverses upwards among the communication layers.

When fuzzing an IoT network stack, a practical solution is typically to use a *native* platform provided by the IoT OS. Most operating systems for resource-constrained IoT devices provide such a platform, including Contiki-NG, RIOT, and Zephyr. The native platform runs as an application in a host OS such as Linux or FreeBSD, and relies on standard POSIX functionality to implement the platform-dependent APIs of the IoT OS. For example, a many of the protocols and applications in the aforementioned IoT operating systems can typically be used in the native mode, except for some protocols below the IPv6 layer, such as the IEEE 802.15.4 Time-Slotted Channel Hopping (TSCH) protocol. At the time of writing, RIOT and Zephyr provide basic fuzzing modules in their respective code repositories, whereas Contiki-NG has fuzzing support through an external fuzzing benchmark repository [31].

While the case study to evaluate NESFuzzer on a real-world system is conducted on the open-source Contiki-NG OS [27], NESFuzzer is not limited to a single OS. To fuzz test another OS such as Zephyr or RIOT using NESFuzzer, one needs to create a fuzzing harness tailored for its application programming interfaces. These modifications entail primarily the insertion of hook function calls to the NESFuzzer library in the input and output functions of the MAC layer or the network layer, but also different changes to avoid using real-time timers and non-deterministic protocol mechanisms. In Section 4, we explain the OS-dependent functionality needed to support fuzzing of Contiki-NG with NESFuzzer.

2.2.2 The Contiki-NG Operating System. As our case study revolves around Contiki-NG, we also give a brief background of this OS. Contiki-NG is designed for resource-constrained IoT devices, with RAM typically in the range of 10-100 kB, and ROM space in the range of 100-1000 kB. Contiki-NG was forked from its predecessor, the Contiki operating system [13], in 2017. Its main principles are to focus on a well-maintained set of IoT platforms, a low-power IPv6 network stack composed of standard protocols, and a predictable release schedule. The user community includes both industry and academia, with Contiki-NG being used in commercial IoT products and as a platform for research [27].

Due to the need to operate in highly resource-constrained devices, Contiki-NG is designed as a monolithic system, where both the core system and applications typically run in the same memory space with limited memory protection and isolation. Additionally, its process scheduling is event-based and cooperative. A vulnerability that can be triggered in any part of the system by an incoming packet can thus have a major effect. Hence, the packet processing paths of all protocols implementations—both in the core system and applications—need to be thoroughly tested.

3 NESFuzzer Design

To enable stateful fuzzing of IoT network stacks, we present the design and implementation of NESFuzzer. When fuzzing a network stack, it is imperative to cover a plethora of protocol implementations operating at different layers. These implementations can be reachable by injecting packets at one or more *entry points* in the SUT. Unfortunately, using current fuzzers without a complex harness limits the attainable coverage due to the state-dependent functionality in certain parts of the network stack. The execution paths taken in a protocol implementation for an input packet can depend on state at lower layers. For instance, when a TCP-based application is running at the top layer, we may need to initialize not only the TCP state but also routing protocol state and IPv6 neighbor discovery state to extend the code coverage.

As shown in Figure 2, NESFuzzer is built with a modular architecture that decouples the tasks of a) generating the state in the network stack and b) performing the fuzzing. This decoupling allows NESFuzzer to be used with a plethora of fuzzing tools, which can be replaced with low effort as the state of the art progresses. NESFuzzer comprises the following key components:

- The state controller, which is a network server process containing a set of programmable protocol state models to generate state. It is responsible for communicating with the harness for a brief period to generate a user-defined state.

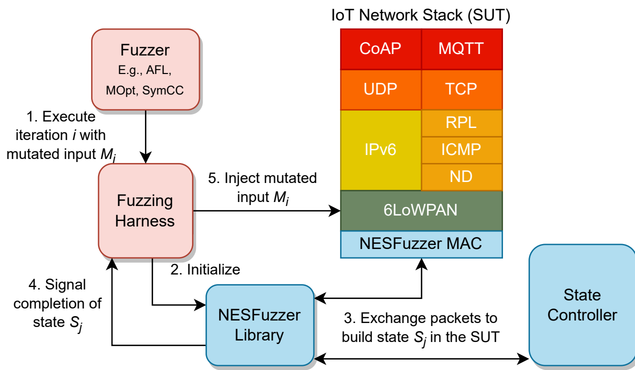

Figure 2: Schematic overview of NESFuzzer in a single fuzzing iteration with state generation. Unlike a typical fuzzing operation, which contains only steps 1 and 5, NESFuzzer inserts steps 2-4 to generate a pre-selected state in the SUT before injecting the fuzzed input.

- A fuzzing harness that injects packets into the IoT network stack. It uses the functionality of the NESFuzzer library to generate protocol states before injecting fuzzed packets.
- The NESFuzzer library, which is linked to the fuzzing harness, and manages the communication between the harness and the state controller to generate state for various protocol implementations in the SUT.
- A common fuzzing tool—such as AFL [39], MOpt [23], or SymCC combined with AFL [30]—that is responsible for executing the SUT through the harness repeatedly, providing mutated input data, and monitoring progress.

3.1 Fuzzing Procedure

When using NESFuzzer, the procedure differs from conventional fuzzing in that the fuzzing harness for the SUT executes a protocol state generation phase. Figure 2 shows the sequence of actions performed during a single fuzzing iteration when generating a specific state in the SUT. The following five steps are taken.

(1) The fuzzing tool executes the harness, providing a mutated input M_i to inject into the SUT.
(2) The harness initializes the NESFuzzer library, providing information such as the destination address and port of the state controller and the selected log method.
(3) The NESFuzzer library forwards packets between the SUT and the state controller to generate a selected state S_j in the SUT. During a fuzzing experiment, the selected state can be switched at times according to one of the policies described in Section 3.4.
(4) The NESFuzzer library signals the completion of the state generation to the harness.
(5) The harness finally injects the mutated input M_i to the SUT.

This differs from a typical fuzzing session, which would have only the aforementioned steps 1 and 5. We call the introduced steps 2-4 the *Protocol State Generation* phase, and describe it in further detail below.

3.2 Protocol State Generation Phase

In order to enable stateful fuzzing of the SUT, the fuzzing harness communicates with the state controller to exchange data packets for different protocols from the 6LoWPAN layer and upwards. Immediately after the fuzzing harness has been initialized in a new fuzzing iteration, we execute the protocol state generation functionality using the NESFuzzer library. This functionality opens a communication session and encapsulates the exchanged data packets using the NESFuzzer Protocol described below.

A desired *end-state* for each protocol to fuzz test is specified in the configuration of the state controller. During the initialization phase, several protocols can communicate simultaneously with the state controller to generate state. Once the configured end-state has been reached for all activated protocols, the state controller terminates the communication session to allow the fuzzing harness to start injecting input data provided by a fuzzer into the SUT.

If a state cannot be generated within a configurable timeout period, the harness exits and lets the fuzzing tool restart the SUT. If such a problem persists over multiple iterations, it could be a configuration error that should be investigated. Yet, in the regular case, the state generation will succeed within the timeout period.

3.2.1 *NESFuzzer Protocol.*

The communication between the fuzzing harness and the state controller is conducted using the NESFuzzer Protocol, a UDP-based protocol that is implemented using Google's Protocol Buffers. The NESFuzzer protocol's main purposes are (1) to signal the start and end of a state generation session, and (2) to carry packets between the network stack and the state controller.

The NESFuzzer Protocol contains three main message types (`START_SESSION`, `END_SESSION`, `PKT`), along with two error message types (`TIMEOUT`, `FAILURE`). The `START_SESSION` message delineates a new state generation phase, which ensures that all protocol state models in the state controller get reset. The `END_SESSION` message communicates to the fuzzing harness that a set of configurable state goals (see Section 3.3) have been reached successfully, whereby the next step is to pass the mutated input from the fuzzing tool into the IoT network stack. The `PKT` message encapsulates full IPv6 or 6LoWPAN messages, which are packed and unpacked at each side of the communication between the NESFuzzer library and the state controller. Furthermore, each sent message has a sequence number and a boolean flag that determines whether the message is (1) initiated by the sender, in which case the sequence number must be strictly monotonically increasing, or (2) a response to a message sent by the other side, in which case the sequence number must match the one of the original message.

3.2.2 *Example State Generation Session.*

Figure 3 shows a basic session consisting of a sequence of packets exchanged to generate two protocol states for RPL and TCP respectively. At the SUT side, the IoT network stack contains a MAC Emulation Layer that uses the NESFuzzer library to forward packets back and forth with the state controller. In this example, the state generation phase ends when the state controller has received one TCP packet with application data in the `ESTABLISHED` state. Additionally, we generate a basic routing state with RPL by sending a DIO message that sets up an initial route with a global IPv6 address prefix in the SUT. This exchange unlocks state-dependent execution paths in the network

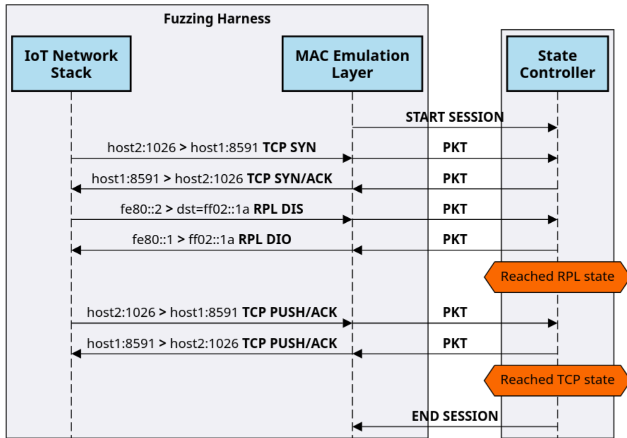

Figure 3: An example session between the fuzzing harness and the state controller to generate two protocol states using interleaved messages. The state is reconstructed before the fuzzing input is inserted into the network stack.

```yaml
protocols:
  tcp:
    reset_policy: reconnect
    listen_port: 8591
    connect_port: 8592
    explore_states: [SYN_RCV, EST, FIN_WAIT_1]
  rpl:
    dodag_root: true
    instance_id: 1
    prefix: fd00::/48
    explore_states: [DIO_SENT, DAO_SENT]
  ipv6_reass:
    packet_size: 1000
    fragment_size: randomize
```

Listing 1: A simple NESFuzzer configuration example in YAML format, with protocol states set for TCP, RPL, and IPv6 fragmentation.

stack that could not have been exercised with fuzzed input packets alone. In this example, these code paths involve packet forwarding, RPL message processing, and TCP packet processing.

3.3 Protocol State Models

The state controller contains a set of *protocol state models*, which provide state generation capabilities for communication protocols. A protocol state model typically implements a simplified state machine that is sufficient to reach a set of state-dependent code paths at various depths in the SUT, as well as the ability to construct and parse the protocol messages needed for this state machine. We use the Scapy library [33] for Python to construct and parse messages for the supported protocols. When implementing a new protocol state model, one can thus leverage Scapy's vast protocol support, and focus on the implementation of the state machine. For instance, our TCP model is implemented in just 258 source lines of code.

Each protocol state model has an *initialization* function that is called once at the start of the state controller, a *reset-state* function that is called for each new state generation session, and a *packet-input* function that is called when receiving traffic from the harness in the protocol state generation phase. If an input packet triggers a response by the protocol state model, the response packet is returned from the input function and sent to the NESFuzzer library where it is encapsulated into another type of packet and transmitted to the state controller.

Before a fuzzing session, one can configure the protocol states that should be generated. Listing 1 shows a configuration example in the YAML format. Currently, NESFuzzer supports four protocol state models for widely used protocols in IoT network stacks: RPL, TCP, 6LoWPAN fragmentation, and IPv6 fragmentation.

3.3.1 RPL State. RPL, the IPv6 Routing Protocol for Low-Power and Lossy Networks [14], is a common state-dependent protocol used in IoT network stacks. Therefore, we implement an RPL protocol state model that can set up various routing states. RPL uses ICMPv6 to exchange control messages, and builds a routing topology in the form of a destination-oriented directed acyclic graph (DODAG). At the start of the SUT, RPL typically sends a DODAG Information Solicitation (DIS) message to ask for information about the DODAG in use by its neighbor nodes, In the state controller, we generate a DODAG Information Object (DIO) containing the necessary information for the SUT to join the routing topology.

The RPL protocol state model can be configured with a variety of settings, including the DODAG ID, the IPv6 address prefix used for auto-configuration, and the *mode of operation* that indicates whether storing or non-storing mode is used [14]. These values should be set to match any expectations of the SUT to prevent the packets from being dropped. In the case of Contiki-NG, the suitable mode of operation depends on whether the RPL-Classic or the RPL-Lite implementation is used, because the latter supports non-storing mode only.

3.3.2 TCP State. We build a simplified implementation of TCP in the state controller. This implementation manages a set of connections identified by the IPv6 address and port number of the source and the destination endpoints. With each connection, we associate a Transmission Control Block (TCB) [37], as well as a few internal variables, such as the application state associated with the TCP connection. The TCP implementation can either connect actively to a remote endpoint or accept incoming connections. When receiving TCP packets, the protocol state model validates the TCP header fields and transitions between different TCP states; e.g., `LISTEN` → `SYN_RECEIVED` → `ESTABLISHED`.

At the harness side, a small TCP socket application is set up to communicate with the TCP protocol state model in the state controller on the configured TCP ports. Observe that this connection does not use the host operating system's socket API, but rather goes through the network stack down to the MAC layer, where the generated packets are intercepted by the NESFuzzer library and forwarded to the state controller, as explained above.

3.3.3 6LoWPAN State. 6LoWPAN is a standard adaptation layer between the IPv6 layer and the IEEE 802.15.4 MAC layer [26]. It can keep state for packet fragmentation and packet compression.

For each fuzzing iteration, NESFuzzer can generate 6LoWPAN fragmentation states according to a configuration that determines a set of injected packet fragments. The fragments can have variable identifiers (i.e., sender address, receiver address, and packet tag ID) in addition to different sizes. For the packet compression functionality, 6LoWPAN instead relies on shared network state when encoding packets in the `LOWPAN_IPHC` format. Because this state is pre-configured rather than generated, NESFuzzer is mainly concerned with the 6LoWPAN fragmentation state.

3.3.4 IPv6 Fragmentation State. The μIP implementation of IPv6 supports fragmentation, which occurs when an IPv6 packet is larger than the maximum transmission unit (MTU) for a given network interface. In NESFuzzer, the IPv6 fragmentation protocol state model operates with one-way traffic directed from the state controller to the harness. The injected packets comprise multiple fragments of configurable sizes. By injecting these fragments into the network stack, a single fuzzed input packet can thereafter exercise the IPv6 fragment reassembly functionality in μIP.

Since the fragments are not sent as replies to any particular packet coming from the network stack, we have added the functionality in the state controller to schedule transmissions of packets at the end of each state generation session. Hence, these packets generate state by relying on the domain knowledge inherent in the protocol state module for IPv6 fragmentation to generate a state in the SUT without a two-way packet exchange to confirm that the state has been reached. In Listing 1, this is configured in the *ipv6_reass* section, where the packet size and the fragment size can be chosen. Unlike the other protocol state models, this section does not contain an *explore_states* keyword because the state generation is limited to simply sending the aforementioned fragments.

3.4 State Selection Policies

Each protocol state model has a predetermined set of states that can be configured to be generated for the SUT during a fuzzing session. During a fuzzing session, the operator configures which of these states should be used, and which policy should be used to switch among them. NESFuzzer supports the following three policies for state generation.

- *Constant state.* The operator configures a specific state to be generated for all iterations.
- *Time-based state switching.* The state is switched automatically after a configured period of time.
- *Progress-based state switching.* A script monitors the fuzzer's progress in a given state. Once the fuzzer has not found a new path in a certain amount of time, it signals state controller to switch state.

Whenever the state is switched, it is logged so that the fuzzing operator can track the state that was generated if a crashing or hanging input is found at some point.

3.5 Test Seed Generation

The test seed is a set of files that fuzzers use to generate inputs for the SUT. The seed selection can have a major impact on the fuzzing results [19]. Still, it can be a challenge to create a suitable test seed, as the typical procedure is to select samples of input data considered representative for the SUT. This problem is particularly evident when fuzzing a network stack, since a packet may have to pass through the packet header validations at multiple network layers. In the case of fuzzing a TCP implementation, packets may need to be injected at the IPv6 layer or below. Thus, in order to reach the TCP code, a packet must have an acceptable IPv6 header; i.e., acceptable source and destination addresses, correct extension headers, and a final next-header value set to 6 (for TCP). Following this header, there must be an acceptable TCP header, with source and destination ports matching those of an existing socket. If one simply collects a packet trace from a network of nodes running the SUT, the captured traffic may come from a session with no error conditions and few protocol extensions being used, thereby containing a limited representation of the protocol's message structure. Additionally, the seed set may tediously need to be re-captured in case the SUT configuration changes.

To address this issue, NESFuzzer provides a simple method to automatically generate a seed set for a specific experiment by recording all the packets exchanged between the harness and the state controller during state generation in a dry-run prior to fuzzing. Hence, this generates a seed set with correct packets for the stateful protocols being fuzzed. If certain protocol parameters get changed, or whole protocols are added or removed, the seed set can be regenerated in a matter of minutes.

4 Network Stack Fuzzing

To demonstrate the benefits of NESFuzzer, we conduct a case study on an IoT network stack. This type of fuzzing target is challenging as it involves multiple state machines for different protocols. In this section, we describe some of the requirements to support such a fuzzing target, including the fuzzing harness and implementation considerations. In Section 5, we use this harness for the evaluation of NESFuzzer. We select the Contiki-NG network stack for this purpose because it contains a plethora of protocol implementations that have a long history of use and have been tested in previous fuzzing efforts [31, 34].

4.1 SUT Requirements and Portability

Although we conduct this case study on a single SUT, we have designed NESFuzzer to be portable to other operating systems for resource-constrained devices with low-power IPv6 network stacks such as Zephyr [3] and RIOT OS [6]. The key requirement from each of these operating systems is that they have a *native platform*, meaning that the system can be compiled and executed as an application running in a larger operating system with POSIX support, such as Linux.

Most of the functionality required to use NESFuzzer is OS-agnostic and can be accessed through the functions of the NESFuzzer library that are shown in Listing 2. This library simplifies the usage of NESFuzzer by hiding the complexity of sending and receiving the packets to the state controller, as well as controlling the state generation process. Hence, the harness-specific part required to support NESFuzzer consists of hooking the packet input and output functions of the host OS into the library module, and adding function calls to initialize the library and to wait for the state to be reached before fuzzing. Meanwhile, the OS-dependent part described below

```
// Set the callback for packets received from the state generator.
void nesfuzzer_set_callback(nesfuzzer_input_callback_t input_cb);
// Start a state-generation session.
bool nesfuzzer_start(const char *nesfuzzer_host,
                     unsigned nesfuzzer_port, bool enable_networking,
                     unsigned log_types, unsigned log_level);
// Stop a state-generation session.
bool nesfuzzer_stop(void);
// Check whether the state-generation phase resulted in an error.
nesfuzzer_error_t nesfuzzer_error(void);
// Send a packet to the state controller.
int nesfuzzer_send(const void *packet_buf, size_t packet_len);
// Check whether the state-building phase finished.
bool nesfuzzer_finished(void);
```

Listing 2: The main functions of the NESFuzzer library application programming interface used in the fuzzing harness.

consists of approximately 700 source lines of code that need to be tailored for a given OS.

4.2 Fuzzing Harness

The fuzzing harness conducts the state generation using the NESFuzzer library, and passes the data generated by a fuzzer to an appropriate input function in the network stack. For our case study, we implement the harness as a Contiki-NG application that is started automatically after the initialization procedure of the operating system. When the harness application starts, it initializes the NESFuzzer library with a set of configuration parameters, such as the protocol entry point to use for fuzzed packets, and the host and port of NESFuzzer's state controller. After waiting for the protocol state generation phase to finish, it reads a fuzzed packet from a file supplied by the fuzzer into a packet buffer.

The harness contains a dispatcher that can inject fuzzed packets into different protocol implementations. When testing a specific protocol implementation, it can be more efficient to inject packets directly into their input handlers and bypass the consistency checks made on headers processed by lower layers. Several protocol implementations, however, depend on functionality at lower layers to operate. We select the IPv6 input function as the main entry point, since IPv6 is the *narrow waist* of the IoT network stack [17], from where the fuzzed input packets can reach the protocols at upper layers. Alternatively, one can inject packets with 6LoWPAN as an entry point, reaching the same set of protocols as long as the packet passes the 6LoWPAN header validation.

4.3 OS-Specific Functionality

In addition to the implementation of fuzzing harness, we need to enable a fast exchange of packets in the protocol state generation phase before the fuzzing harness can inject a packet to the network stack in a specified state. In the following, we describe various aspects of the Contiki-NG system that need to be considered, and the minor modifications that we make to the system internals.

4.3.1 MAC Emulation Layer. To be able to exchange packets in the protocol state generation phase, we replace Contiki-NG's medium access control (MAC) driver with a MAC emulation driver, as is shown in Figure 2. The emulated MAC driver intercepts packets transmitted downward from the IPv6 layer, and injects incoming packets from the state controller into the IPv6 layer.

Instead of exchanging data with an underlying link layer, the emulated MAC driver sets up a callback in the NESFuzzer library to receive IPv6 packets from NESFuzzer and propagate them up through the network stack with the IPv6 layer as the entry point. Conversely, outgoing packets are received by the MAC driver from the IPv6 module, and then forwarded to the NESFuzzer library by a single call to the *nesfuzzer_send* function.

4.3.2 Time Management. Although we execute Contiki-NG as a native process in an OS such as Linux, it performs all of the regular operations with respect to process scheduling and timer management. As several protocols rely on the timer functionality, this can have implications on how fast the protocol states can be generated. For example, the RPL protocol uses exponentially increasing timers for DIO messages, and configurable intervals for other types of control messages.

Since it would be prohibitively slow to wait for these timers to expire, we modify Contiki-NG to run with an accelerated time scale. This can be done by simply modifying a statement in the `os/sys/timer.c` module that sets the time interval in the `timer` data structure. There is a trade-off, however, in configuring too fast a time-scale, because certain protocols such as RPL might generate maintenance traffic, and thus prolong the sequence of messages sent between the harness and the state controller before the end-state for all active protocols is reached. Hence, the time scale may need to be fine-tuned when fuzzing a given network stack configuration. Still, our preliminary experiments have shown that a timer speed of 100 times faster than real time is a suitable default setting for our Contiki-NG implementation.

4.3.3 Mitigating Non-Determinism. Both the execution of NESFuzzer and the SUT can be subject to non-determinism, which can be observed in the *stability* metric in AFL-based fuzzers. In other words, two identical inputs to the SUT can lead to different paths being taken when executing it at different times. Several factors in the computing environment can contribute to non-determinism. For instance, the network-based architecture of NESFuzzer is dependent on the packet processing times in the host operating system. If the timing varies, the number of packets exchanged for state generation can vary as well. Another factor can be the built-in randomness in the protocols.

We mitigate these issues by 1) setting the SUT's random seed to the same value for each execution, 2) increasing RPL's periodic timers to long intervals so as to remove redundant packets during the state generation phase, and 3) disabling active link probing in RPL, which is used to discover neighbor nodes and update their link metrics. These actions reduce the traffic to the state controller that does not generate interesting states for fuzzing. When using AFL-based fuzzers, the harness also resets the coverage data when the state-generation phase has finished. This ensures that the coverage data is not affected by the non-determinism of the packet processing times in the host OS during the state-generation phase.

5 Evaluation

We experimentally evaluate NESFuzzer with respect to its achieved code coverage, test seed efficiency, and execution performance. Moreover, we demonstrate NESFuzzer's capability to enhance the vulnerability discovery in IoT network stacks. As a result of this work, we present three zero-day vulnerabilities that had existed for several years in Contiki-NG's network stack, and which could not be discovered with stateless fuzzing in an extensive benchmark of state-of-the-art fuzzers [31]. In addition to finding these vulnerabilities, we demonstrate that NESFuzzer can find a known state-dependent vulnerability in an earlier version of Contiki-NG's TCP implementation.

5.1 Experimental Setup

We conduct the experiments on a machine with an Intel Xeon W-1370P CPU with 64 GB RAM running Ubuntu Linux 22. We use the following tools together with NESFuzzer in our experiments.

(1) *American Fuzzy Lop* (AFL) 2.57b, which is a widely used coverage-guided, mutation-based fuzzer.
(2) *SymCC* (Git commit version `9b20609ada`), which is a symbolic execution tool that can be combined with an AFL-based fuzzer to form a hybrid fuzzer.
(3) *MOpt* (Git commit version `a9a5dc5c0c`), a fuzzer based on AFL that uses a particle swarm optimization (PSO) technique to schedule mutations more efficiently.

In our experiments, we use Contiki-NG version 4.9 in its *native* platform mode as the SUT. The network stack is the same at the source-level from the 6LoWPAN layer and upwards, as when running a Contiki-NG firmware on an IoT platform. For the state controller, we enable three different protocol state models to build state in various parts of the network stack: IPv6 reassembly, RPL, and TCP. To reduce the interpretation overhead, we execute the state controller using PyPy version 7.3.1, which provides a just-in-time compiler for Python software. Moreover, each fuzzer instance has a dedicated state controller process.

5.2 Code Coverage

To evaluate the coverage achieved with NESFuzzer, we conduct a set of experiments that measure the edge coverage of the SUT, in addition to inspecting the branch, line, and function coverage in the network stack modules in more detail.

5.2.1 Network Stack Coverage. We first examine the network stack coverage using the aggregated output corpora generated by the fuzzing tools during 24-hour runs for each setting. For this purpose, we build a script to compare the coverage of two fuzzing experiments, *Exp1* and *Exp2*, as obtained from the *afl-cov* tool. For each file in the network stack, we print the differences between the sets of lines and functions covered in *Exp1* and *Exp2*.

Figure 4 shows the attained coverage in key modules of Contiki-NG's network stack with NESFuzzer compared to using SymCC alone. Major coverage increases can be observed both when using MOpt and SymCC with NESFuzzer. The increases are over 170% in certain modules of the RPL implementation, where large areas of the code cannot be reached with fuzzed input packets without setting up certain states. The `uip` and `tcpip` modules, which contain the

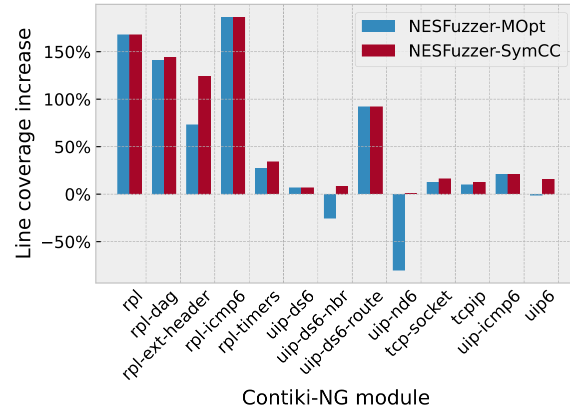

Figure 4: Increase in line coverage when fuzz testing the network stack in Contiki-NG with NESFuzzer. We increase the coverage when using NESFuzzer-MOpt and NESFuzzer-SymCC compared with that of regular SymCC fuzzing.

core IPv6, ICMPv6, TCP, and UDP functionality, gain less because major parts of their code are stateless. Still, the gain is approximately 13% for `uip` and 17% for `tcpip` when executing NESFuzzer with SymCC. For modules that contain extensive sections that do not depend on state, the coverage can decrease with stateful fuzzing due to the lower execution speed. We note, however, that NESFuzzer-SymCC increases the coverage for all of the tested modules.

Table 1 shows the aggregated coverage attained from the output corpora when fuzzing with SymCC and MOpt combined with NESFuzzer. The lighter green color indicates advantages up to five percentage points, whereas the darker green color indicates advantages above five percentage points. NESFuzzer manages to reach state-dependent code paths, including different packet fragmentation states, routing protocol states, and TCP socket states. The table shows that the line coverage surpasses 80% in several of the main modules of the network stack, including the core `uip6` module, which contains a large part of the IPv6 implementation.

Still, even with the stateful fuzzing of NESFuzzer, it should be noted that is difficult to reach a 100% coverage for two main reasons. 1) Certain code paths cannot be reached without inserting statements in the harness that are independent of packet fuzzing. For example, these could be paths that are executed when allocating the maximum number of sockets or closing an already closed socket. It is possible that a complementary fuzzing method that injects sequences of system calls—such as the one provided by Syzkaller [2]—can further increase the coverage. 2) We are not generating an exhaustive set of states for the protocol implementations. To further enhance the state-generation, it would be interesting for future work to investigate how one can integrate NESFuzzer with state machine learning methods for protocols [10].

5.2.2 Coverage Map Density. AFL keeps an approximate track of edge coverage in the instrumented SUT through a shared memory bitmap that is typically 64 kB. The coverage map density denotes

Table 1: Coverage achieved when using NESFuzzer together with either SymCC (denoted as $SymCC_{NF}$) or MOpt (denoted as $MOpt_{NF}$). Green color indicates a major improvement in coverage for a specific module, whereas light green indicates a minor improvement.

Module	Lines (%)		Functions (%)		Branches (%)	
	$SymCC_{NF}$	$MOpt_{NF}$	$SymCC_{NF}$	$MOpt_{NF}$	$SymCC_{NF}$	$MOpt_{NF}$
rpl	36.4	36.4	35.3	35.3	27.8	27.8
rpl-dag	58.2	58.2	51.9	51.9	37.7	37.5
rpl-ext-header	85.5	86.3	100.0	100.0	64.3	67.9
rpl-icmp6	86.5	87.1	85.7	85.7	59.3	63.2
rpl-timers	56.8	56.8	35.3	35.3	27.8	27.8
uip-ds6	76.1	76.1	60.0	60.0	50.0	50.0
uip-ds6-nbr	78.5	78.5	57.1	57.1	57.7	57.7
uip-ds6-route	76.1	57.6	50.0	50.0	30.9	30.9
uip-nd6	98.0	53.5	100.0	66.7	60.8	28.4
simple-udp	74.0	44.0	60.0	40.0	48.3	27.6
tcp-socket	69.4	69.4	62.5	62.5	44.6	43.8
tcpip	64.8	63.2	72.7	72.7	60.5	48.8
uip-icmp6	90.3	90.3	80.0	80.0	82.0	80.0
uip6	80.1	70.4	83.3	83.3	68.4	61.4
uipbuf	69.8	65.1	50.0	50.0	59.1	54.5

the percentage of bytes set in the bitmap for the experiment's entire corpus of inputs. Although this metric does not represent the percentage of possible paths covered, it can be used to compare the attained coverage of two different fuzzing experiments for the same SUT binary.

We compare the coverage map density of stateless SymCC fuzzing with that of NESFuzzer combined with either SymCC or MOpt. We denote these configurations as NESFuzzer-SymCC and NESFuzzer-MOpt, respectively. For both of the SymCC experiments, we use one AFL coordinator and one worker process. Since MOpt is an extension of AFL, we use one MOpt-AFL controller and two MOpt-AFL worker processes to compensate for the extra process that the SymCC setup uses.

Figure 5 shows the arithmetic mean of the coverage map density achieved over time when repeating the experiment ten times. The shaded areas show 90% confidence intervals. We find that NESFuzzer-SymCC achieves a considerably higher map density compared to NESFuzzer-MOpt and regular SymCC fuzzing. SymCC in stateless mode achieves a coverage close to that of NESFuzzer-MOpt. We attribute the map density achieved by SymCC in the stateless mode to its efficient symbolic execution, and the considerably faster execution of stateless fuzzing. Still, a key difference between the results of NESFuzzer-MOpt and stateless SymCC is that the sets of covered paths differ, as shown in Figure 4. Although the stateful fuzzing is still covering stateless code blocks, SymCC in the stateless mode can cover more such code blocks because it can execute many more fuzzing iterations in the same time. By contrast, both the NESFuzzer-MOpt and NESFuzzer-SymCC fuzzers cover stateful code blocks that cannot be covered by SymCC alone.

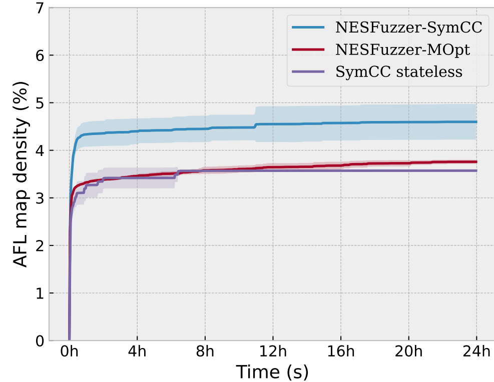

Figure 5: Timeline of attained coverage map density when fuzzing Contiki-NG, showing the arithmetic mean and 90% confidence intervals for each setup. NESFuzzer-SymCC achieves the highest coverage leveraging both stateful and hybrid fuzzing.

5.3 Impact of Automatic Seed Generation

We also examine how the test seeds generated from NESFuzzer affect the fuzz testing coverage. In particular, we compare two different test seeds: 1) a seed consisting of a set of IPv6 packets extracted from a network of Contiki-NG nodes, and 2) a seed automatically generated by recording a dry-run from NESFuzzer when generating states of various protocols. For each seed set, we run a 24-hour fuzzing session with MOpt and NESFuzzer, and collect path coverage data for the Contiki-NG network stack.

Figure 6 shows the timeline of two 24-hour experiments with NESFuzzer-MOpt using two different seed sets. The first one consists of IPv6 packets extracted from PCAP files in a simulated Contiki-NG network. The second one is generated by NESFuzzer as explained above. We find that the generated seed set yields a considerably higher path coverage over time. Initially, both experiments quickly discover a majority of the paths that are reached over the full time, but with the plain IPv6 seed, the fuzzer struggles to find certain code areas through random mutations after about an hour has passed. By contrast, the NESFuzzer-generated seed helps the fuzzer to make progress for a longer time using the set of correctly formed packets for the protocol states configured for the

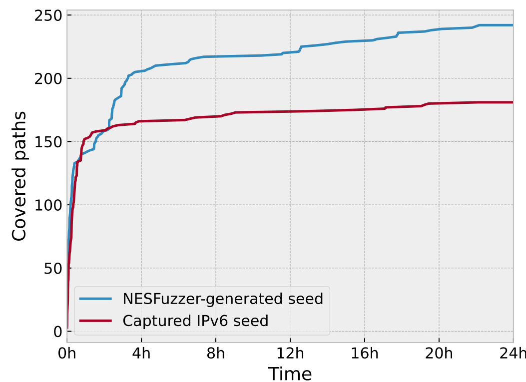

Figure 6: Path coverage attained over time when fuzzing with NESFuzzer-MOpt using a stateless IPv6 seed compared to using a NESFuzzer-generated seed. The seed set generated by NESFuzzer results in a considerably higher path coverage.

experiment. After 24 hours had passed, the NESFuzzer-generated seed covered approximately 34% more paths.

5.4 Execution Performance

We compare the performance metrics of a) stateless fuzzing with one SymCC process, one AFL controller process, and one AFL worker processes (*Stateless SymCC*), b) stateful fuzzing with MOpt using one controller process and two worker processes similar to AFL (*NESFuzzer-MOpt*), and c) stateful fuzzing with one SymCC process, one AFL controller process, and one AFL worker process (*NESFuzzer-SymCC*). Each experiment is repeated ten times over a duration of ten minutes per repetition, and using the same instrumented executable. We use a considerably shorter duration for these experiments because the execution performance stabilizes quickly after a few seconds in the startup phase. Table 2 shows the execution speed and stability achieved in the experiments.

5.4.1 Execution Speed. We find that the regular SymCC session without state generation yields the fastest execution times, as it does not have the overhead of exchanging packets with the state controller to set up network stack state. While the NESFuzzer-enabled experiments execute considerably slower in AFL's default mode, they are both able to discover three vulnerabilities (see Section 5.5) that the Stateless SymCC alone could not find. Although a faster execution speed ensures that more inputs can be injected into the SUT, it might not lead to a higher coverage compared to what a slower, but more effective fuzzer can achieve. In particular, this is the case when the SUT has state-dependent code paths. Hence, we trade off speed for the capability to generate states in the SUT. For this reason, it can be beneficial to first test an SUT using stateless fuzzing with a considerably higher execution speed, and then switch to stateful fuzzing once the fuzzer has exhausted its capability to find code paths.

A major improvement of the execution speed is possible in some cases, however. When using an AFL-based fuzzer, one might be able to use its *deferred mode* [39]. In the default AFL mode, it executes the SUT once and then forks this process for each fuzzing iteration. By deferring this fork point to come after the state generation has finished, the child process can resume execution from that point. Hence, as shown in Table 2, one can avoid the state generation overhead if a constant state is set for the fuzzing session. The applicability of this mode depends on the SUT, as the SUT can malfunction if it allocates certain system resources such as timers and threads before the deferred fork point. While we use deferred mode only for constant state settings, it would be interesting for future work to integrate this mode with NESFuzzer functionality use different states within a session.

5.4.2 Stability. Furthermore, we examine the stability of the fuzzing experiments. This metric is a measurement of the consistency in the code paths executed for identical inputs. It is measured in AFL-based fuzzers when a new test case is generated. In Table 2, we see that all three alternatives achieved a stability of over 99.7% during the experiments. Hence, we were able to reduce the non-determinism as described in Section 4.3.3, allowing coverage-guided fuzzers to function properly with the coverage data from the SUT.

5.5 Discovered Vulnerabilities

In our experiments with NESFuzzer on Contiki-NG, we discovered three zero-day vulnerabilities in the RPL and TCP implementations that could not be found with MOpt or SymCC alone. Each of these vulnerabilities have now been fixed in the main Contiki-NG repository. Additionally, we could re-discover the vulnerability described in `CVE-2020-17437` regarding a non-validated TCP urgent pointer value. We briefly describe the three vulnerabilities below.

5.5.1 TCP Offset Processing Vulnerability. During our fuzzing experiments with NESFuzzer, we found a state-dependent vulnerability (`CVE-2021-21281`) that concerns non-validated data in the TCP header. The problem exists in the *uip_process* function in Contiki-NG's `os/net/ipv6/uip6.c` module. The TCP offset field in the TCP header is not verified, making it possible to cause an overflow of the *uip_len* variable. Thereafter, the execution can reach the `os/net/ipv6/tcp-socket.c` module, where the TCP payload is copied into a user-supplied buffer. The amount of data copied is determined by the value in the overflowed *uip_len* variable, and can therefore lead to a buffer overflow.

In order to reach the vulnerable code through fuzzing, however, the TCP module must have at least one connection in established state. Additionally, the fuzzer must mutate a packet such that all fields from the IPv6 and the TCP header are correct to prevent the network stack from dropping the packet while processing and validating the header fields.

5.5.2 RPL Target Option Vulnerability. The second state-dependent vulnerability found (`CVE-2021-32771`) with NESFuzzer is in the processing of RPL DAO messages. The bug manifests itself when ContikiRPL operates in storing mode, and has joined an RPL instance. When an incoming DAO packet contains an RPL Target Option, an IPv6 address prefix should be copied from an offset in the packet into a local variable. The prefix length is specified in a different field, but its value is not validated before supplying the value as an argument to the `memcpy` call, thereby making it possible to overflow the buffer and cause Contiki-NG to crash.

Table 2: Run-time performance using stateful fuzzing with either NESFuzzer-MOpt or NESFuzzer-SymCC compared to stateless fuzzing with SymCC. The stateless fuzzing achieves a considerably higher execution speed in the default AFL mode, but this overhead is removed when using AFL's deferred mode to reuse the SUT process state.

	Stateless SymCC	NESFuzzer-MOpt	NESFuzzer-SymCC
Exec/s with default mode	3307 ± 113	38.61 ± 2.68	35.85 ± 0.81
Exec/s with deferred mode	6919 ± 54	5599 ± 120	6854 ± 128
Min. stability	99.87%	99.73%	99.84%

5.5.3 Unaligned Memory Access in RPL Option Processing. Lastly, our stateful fuzzing experiments led to the discovery of an unaligned memory access vulnerability in the processing of RPL options (`CVE-2024-47181`) within an IPv6 extension header. This can enable an attacker to trigger undefined behavior in the system. Once the Contiki-NG device has joined an RPL instance, this vulnerability can be triggered by prepending the RPL option with either a PAD1 or PADN option [11] that inserts an odd number of bytes.

6 Related Work

In the following, we discuss the related literature pertaining to stateful protocol fuzzing and IoT software fuzzing.

6.1 Stateful Protocol Fuzzing

The fuzzing community has identified the need to support fuzzing of network protocol implementations. A primitive way of achieving such fuzzing—at least partially—is to build custom fuzzing harnesses for each SUT that generate state by accessing the SUT's internal data structures, calling functions from its public interfaces, or injecting specific sequences of inputs. Yet, this approach requires intricate knowledge of the SUT to set up the harness, and the manual effort required can be tedious and fail to cover a satisfactory set of states.

AFLNet introduces a more generalized method to enable fuzzing of a variety of network servers over sockets [29]. AFLNet infers protocol states using status codes parsed from protocol messages, and uses AFL's mutation of messages to try to reach new states. On the flip side, it is less suitable for protocols that rely on temporal information or lack status codes. To support a new protocol, one has to implement certain parts of the protocol parsing in order to extract state indicators in AFLNet.

Fuzzer in the Middle (FitM) enables fuzzing of stateful client-server communication by extending AFL's QEMU mode with a network emulation layer [24]. Its network emulation layer allows it to avoid the overhead of system calls used by earlier methods such as AFLNet. To further improve the performance, it uses process state snapshots that can be restored in each iteration rather than regenerating the state through message exchanges. NSFuzz employs static analysis of the SUT to infer state models [32]. It focuses on identifying two parts of protocol implementations: (1) the network event loop that processes input messages, and (2) state variables.

In contrast with these methods, NESFuzzer targets full IoT network stacks rather than single protocol implementations, and enables domain-aware and programmable state generation to test multiple protocols in a single fuzzing session. Furthermore, NESFuzzer decouples the fuzzer from the state generation, enabling it to be used with different fuzzers as the state of the art progresses.

In recent work, Amusuo et al. propose a framework for testing the packet validation functionality of embedded network stacks in different states [4]. Instead of relying on fuzzing to generate input data, they generate correctly structured packets with invalid header field values, and use sanitizer tools to detect errors. The states are generated using an extended version of PacketDrill [8], a tool that supports scriptable packet exchanges for testing network stacks. We view this method as complementary because NESFuzzer is focused on combining different fuzzers with a decoupled state generation mechanism. This allows NESFuzzer to leverage the capabilities of fuzzers, such as mutation strategies, code coverage feedback, and combinations with symbolic execution.

6.2 IoT Software Fuzzing

Various methods have been developed for fuzz testing IoT software, including full firmwares that can be deployed on real hardware. Yet fuzzing IoT software can be challenging because such software typically executes on resource-constrained devices, which reduces the set of applicable tools. Furthermore, there is typically a lack of visibility into the IoT firmware of different vendors, which precludes fuzzing-aware binary instrumentation. For this reason, researchers have proposed blackbox fuzzing methods that rely on messages being sent to IoT devices over the air, and monitoring changes in the internal state of the IoT firmware. IoTFuzzer replays a sequence of captured packets before injecting a fuzzed packet [9]. Alternatively, correctly formed packets can be inferred by mutating bytes one-by-one, sending the data to the SUT and observing the response, as demonstrated by Snipuzz [15]. Zhang et al. use large language models to generate fuzzing inputs for system calls in embedded operating systems [40].

Another possible method for IoT firmware fuzzing is to combine fuzzing tools with hardware emulation. Firm-AFL enables fuzz testing of firmware by a method called *augmented process emulation* [41], which leverages user-mode emulation for efficiency. Scharnowski et al. propose a firmware-aware fuzzer that uses hardware emulation and considers the various input streams that need to be processed by the system [34]. This method supports type-aware mutations for different input streams, as opposed to a single binary input for a chosen fuzzing entry point in the system.

While these methods are effective for discovering bugs in firmware, the execution speed can be hampered by using hardware emulation or sending mutated messages over the air. By contrast, NESFuzzer enables *stateful* and *hybrid* fuzzing of IoT network stacks, but requires an SUT that can execute in native (POSIX) mode. We consider these methods to be complementary to our work because fuzzing the firmware executing on IoT devices can cover code that is unavailable in the native platform, such as a device driver.

7 Conclusions

In this paper, we have designed and implemented NESFuzzer, a system for stateful fuzzing of IoT network stacks. The key idea of NESFuzzer is to decouple the fuzzer from the state generation framework through a network-based architecture. This decoupling enables fuzzing tool operators to leverage a plethora of state-of-the-art fuzzers, while generating different states flexibly for various IoT protocol implementations in embedded network stacks.

As shown in our evaluation using the low-power IPv6 stack of the Contiki-NG operating system as a case study, NESFuzzer increases the code coverage considerably in the modules that implement the network and transport layers. Even in the the slowest mode of re-generating state for each fuzzing iteration, NESFuzzer provides new coverage. Yet, when using an optimization such as AFL's deferred mode, this state generation does not need to take place in each iteration, making the overhead negligible. Through our experiments, we found that the stateful multi-protocol fuzzing method of NESFuzzer enabled us to discover three new security vulnerabilities in state-dependent parts of the TCP and RPL implementations of Contiki-NG. Furthermore, we were able to re-discover a known vulnerability in the TCP implementation. The results demonstrate that NESFuzzer's ability to generate protocol states for fuzzing can improve test coverage and reveal zero-day vulnerabilities.

Acknowledgments

This work was partly funded by the Swedish Foundation for Strategic Research and by Digital Futures.

References

[1] [n. d.]. lwIP - A Lightweight TCP/IP Stack. Web page. https://savannah.nongnu.org/projects/lwip/ Visited 2025-02-18.
[2] [n. d.]. syzkaller. Web page. https://github.com/google/syzkaller Visited 2025-02-18.
[3] [n. d.]. Zephyr Project. Web page. https://zephyrproject.org/ Visited 2025-02-18.
[4] P. C. Amusuo, R. A. Calvo Méndez, Z. Xu, A. Machiry, and J. C. Davis. 2023. Systematically Detecting Packet Validation Vulnerabilities in Embedded Network Stacks. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Kirchberg, Luxembourg.
[5] A. Andronidis and C. Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (Virtual, South Korea).
[6] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. 2018. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet of Things Journal* 5, 6 (2018), 4428–4440.
[7] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. 1995. Improving TCP/IP Performance over Wireless Networks. In *The 1st Annual International Conference on Mobile Computing and Networking (MobiCom)*. Berkeley, CA, USA.
[8] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H. Chu, A. Terzis, and T. Herbert. 2013. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX Annual Technical Conference (USENIX ATC)*.
[9] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. Lau, M. Sun, R. Yang, and K. Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.
[10] J. De Ruiter and E. Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium*. Washington, DC, USA.
[11] S. Deering and R. Hinden. 1998. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. IETF.
[12] D. dos Santos, S. Dashevskyi, J. Wetzels, and A. Amri. 2020. AMNESIA:33. https://www.forescout.com/research-labs/amnesia33/.
[13] A. Dunkels, B. Grönvall, and T. Voigt. 2004. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the IEEE Workshop on Embedded Networked Sensor Systems (Emnets)*. Tampa, FL, USA.
[14] T. Winter (ed.), P. Thubert (ed.), A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander. 2012. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. IETF.
[15] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang. 2021. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *The ACM Conference on Computer and Communications Security (CCS)*. Virtual event, Republic of Korea.
[16] A. Francillon and C. Castelluccia. 2008. Code injection attacks on harvard-architecture devices. In *15th ACM conference on Computer and communications security (CCS)*.
[17] J. Hui and D. Culler. 2008. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*. Raleigh, NC, USA.
[18] H.-S. Kim, S. Kumar, and D. Culler. 2019. Thread/OpenThread: A Compromise in Low-Power Wireless Multihop Network Architecture for the Internet of Things. *IEEE Communications Magazine* 57, 7 (2019), 55–61.
[19] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. 2018. Evaluating Fuzz Testing. In *25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.
[20] J. Ko, A. Terzis, S. Dawson-Haggerty, D. Culler, J. Hui, and P. Levis. 2011. Connecting Low-Power and Lossy Networks to the Internet. *IEEE Communications Magazine* 49, 4 (2011), 96–101.
[21] S. Kumar, M. Andersen, H.-S. Kim, and D. Culler. 2020. Performant TCP for Low-Power Wireless Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Santa Clara, CA, USA.
[22] Y. Li, S. Ji, Y. Chen, S. Liang, W. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium*.
[23] C. Lyu, S. Ji, C. Zhang, Y. Li, W-H. Lee, Y. Song, and R. Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium*.
[24] D. Maier, O. Bittner, M. Munier, and J. Beier. 2022. FitM: Binary-Only Coverage-Guided Fuzzing for Stateful Network Protocols. In *Workshop on Binary Analysis Research (BAR)*. San Diego, CA, USA.
[25] B. P. Miller, L. Fredriksen, and B. So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44.
[26] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. 2007. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. IETF.
[27] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, and N. Tsiftes. 2022. The Contiki-NG open source operating system for next generation IoT devices. *SoftwareX* 18 (2022), 101089.
[28] B. S. Pak. 2012. *Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution*. Master's thesis. School of Computer Science, Carnegie Mellon University. CMU-CS-12-116.
[29] V.-T. Pham, M. Böhme, and M. Roychoudhury. 2020. AFLNet: A Greybox Fuzzer for Network Protocols. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Porto, Portugal.
[30] S. Poeplau and A. Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium*.
[31] C. Poncelet, K. Sagonas, and N. Tsiftes. 2022. So Many Fuzzers, So Little Time - Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Rochester, MI, USA.
[32] S. Qin, F. Hu, B. Zhao, T. Yin, and C. Zhang. 2022. NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing. In *International Fuzzing Workshop (FUZZING)*. San Diego, CA, USA.
[33] R Rohith, M. Moharir, G Shobha, et al. 2018. SCAPY-A powerful interactive packet manipulation program. In *International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*. Bangalore, India.
[34] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz. 2023. HOEDUR: embedded firmware fuzzing using multi-stream inputs. In *32nd USENIX Security Symposium*. Anaheim, CA, USA.
[35] K. Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *26th USENIX Security Symposium*. Vancouver, BC, Canada.
[36] Z. Shelby, K Hartke, and C. Bormann. 2014. *The Constrained Application Protocol (CoAP)*. RFC 7252. IETF.
[37] W. Eddy, Ed. 2022. *Transmission Control Protocol (TCP)*. RFC 9293. IETF.
[38] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister. 2012. OpenWSN: A Standards-Based Low-Power Wireless Development Environment. *Transactions on Emerging Telecommunications Technologies* 23, 5 (2012), 480–493.
[39] M. Zalewski. [n. d.]. American Fuzzy Lop. Web page. http://lcamtuf.coredump.cx/afl/ Visited 2025-02-18.
[40] Q. Zhang, Y. Shen, J. Liu, Y. Xu, H. Shi, Y. Jiang, and W. Chang. 2024. ECG: Augmenting Embedded Operating System Fuzzing via LLM-Based Corpus Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 4238–4249.
[41] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium*. Santa Clara, CA, USA.