Systematically Detecting Packet Validation Vulnerabilities in Embedded Network Stacks

Paschal C. Amusuo
Electrical and Computer Engineering
Purdue University
West Lafayette, USA
pamusuo@purdue.edu

Ricardo Andrés Calvo Méndez
Systems and Computer Engineering
Universidad Nacional de Colombia
Bogotá, Colombia
rcalvom@unal.edu.co

Zhongwei Xu
Systems and Computer Engineering
Xi'an JiaoTong University
Xi'an Shaanxi, China
2206515211@stu.xjtu.edu.cn

Aravind Machiry
Electrical and Computer Engineering
Purdue University
West Lafayette, USA
amachiry@purdue.edu

James C. Davis
Electrical and Computer Engineering
Purdue University
West Lafayette, USA
davisjam@purdue.edu

***Abstract*—Embedded Network Stacks (ENS) enable low-resource devices to communicate with the outside world, facilitating the development of Internet of Things and Cyber-Physical Systems. Some defects in ENS are thus high-severity cybersecurity vulnerabilities: they are remotely triggerable and can impact the physical world. While prior research has shed light on the characteristics of defects in many classes of software systems, no study has described the properties of ENS defects nor identified a systematic technique to expose them. The most common automated approach to detecting ENS defects is feedback-driven randomized dynamic analysis ("fuzzing"), a costly and unpredictable technique.**

This paper provides the first systematic characterization of cybersecurity vulnerabilities in ENS. We analyzed 61 vulnerabilities across 6 open-source ENS. Most of these ENS defects are concentrated in the transport and network layers of the network stack, require reaching different states in the network protocol, and can be triggered by only 1-2 modifications to a single packet. We therefore propose a novel systematic testing framework that focuses on the transport and network layers, uses seeds that cover a network protocol's states, and systematically modifies packet fields. We evaluate this framework on 4 ENS and replicated 12 of the 14 reported IP/TCP/UDP vulnerabilities. On recent versions of these ENSs, it discovered 7 novel defects (6 assigned CVES) during a bounded systematic test that covered all protocol states and made up to 3 modifications per packet. We found defects in 3 of the 4 ENS we tested that had not been found by prior fuzzing research. Our results suggest that fuzzing should be deferred until after systematic testing is employed.

***Index Terms*—Automated Testing, Validation, Cybersecurity, Embedded systems, IoT, Networking, Empirical Software Engineering, Fuzzing**

I. Introduction

Embedded Network Stacks (ENSs) are software components that enable network communication on embedded systems. There are several ENSs with varied architectures tailored to the semantics of individual embedded operating systems, such as Contiki-ng [1] and FreeRTOS [2]. Unlike the network stacks used by regular operating systems, ENSs run on embedded systems with limited or no vulnerability protections [3]. As a result, vulnerabilities in ENSs are severe and could be remotely exploitable. In the last five years, many critical defects have been discovered and reported in these ENSs [4]–[6]. Detecting cybersecurity vulnerabilities in ENSs remains an important challenge for securing the Internet of Things.

Automated software testing techniques for network stacks use formal methods, static analysis, and dynamic analysis to detect vulnerabilities. Formal methods, *e.g.,* model checking, provide strong guarantees [7]–[12] but are costly to apply and maintain. Static analyses efficiently find defects [13]–[17], but must be tuned to defect patterns and generate false positives. Dynamic analysis is promising, especially fuzzing [18]–[22]. But while fuzzing ensures no false positives, it offers limited guarantees. No dynamic works examine the systematic testing of ENSs and consequently provide guarantees.

Our goal was to develop a systematic dynamic testing technique, one that could provide certain guarantees about the security of the ENS under test. But what guarantees should be prioritized? Several studies [23]–[25] show that defect patterns recur in software. Thus, identifying the characteristic defects can help prevent such defects in the future. We analyzed 61 security defects that were previously reported across 6 embedded network stacks to understand defect patterns. We found that most ENSs vulnerabilities occur because an ENS directly used certain fields in packet headers without proper validation. Invalid values of such fields lead to out-of-bound (OOB) reads, buffer overflows, and integer wraparounds. We call these **packet validation vulnerabilities**. Our study also revealed that the test suites used in ENSs are inadequate to detect this recurring class of defect. To dynamically detect packet validation vulnerabilities, an approach must be systematic in varying fields (many different packet fields were problematic), able to reach different protocol states (many different protocol states were problematic), and able to find memory errors (most vulnerabilities involve OOB memory access).

Based on this analysis, we propose EmNetTest, an auto-

mated and systematic framework for dynamic testing of ENSs. EmNetTest possesses three characteristics that enable it to uncover known vulnerability patterns in ENSs. *(1) Systematic packet generation:* EmNetTest systematically generates validly constructed packets with invalid header fields or truncated headers. *(2) Stateful:* EmNetTest provides sequences of packets that get the ENS to different protocol states before packet injection. *(3) Memory focused:* EmNetTest uses address sanitizers with dynamic memory poisoning to detect all memory corruptions. We implemented EmNetTest using PACKETDRILL which provides necessary scripting support for testing network stacks. We enhanced PACKETDRILL to support mutating arbitrary network packets.

We evaluated EmNetTest on 4 of the 6 ENSs whose vulnerabilities we studied: FreeRTOS, Contiki-ng, lwIP, and PicoTCP. We also created ENSBench, a dataset of 12 vulnerabilities by re-introducing previously known vulnerabilities into recent versions of the ENSs. Our evaluation showed that EmNetTest replicated all the 12 vulnerabilities we attempted. In addition, EmNetTest found 7 new vulnerabilities (zero days), which can be remotely exploited by any user and potentially allow arbitrary code execution. We compared our framework with fuzzing. We ran 4 fuzzers from the Poncelet *et al.* benchmarks [22] on the latest version of Contiki-ng (which contains 5 vulnerabilities) and found that within 24 hours, no fuzzer detected any of the vulnerabilities.

Our work shows the importance and effectiveness of systematic testing for detecting critical software defects. We invite the community to explore systematic testing approaches, beyond the current trend of automated randomized testing (fuzzing).

In summary, we contribute:

1) We perform the first comprehensive study (§V) of 61 reported ENS vulnerabilities, understand their root causes, and provide insights into the packet sequences that trigger these vulnerabilities.
2) We designed and implemented EmNetTest (§VI), an automated systematic testing framework for ENS. Our evaluation shows that EmNetTest effectively finds known and new vulnerabilities in ENS.
3) As part of our framework, we implemented PACKETDRILL++, an extended version of PACKETDRILL that facilitates adversarial testing of network stacks and can be used independently of our testing framework.
4) ENSBENCH: A dataset of 12 recreated and 7 new vulnerabilities, packaged into recent versions of ENSs to support the evaluation of other defect detection tools. EmNetTest detects all vulnerabilities in this dataset.

Our vulnerability analysis and the implementation of EmNetTest are available (§XI).

II. BACKGROUND

A. Embedded Network Stacks (ENS)

Embedded Network Stacks (ENSs) enable network connectivity for embedded systems. ENSs are either part of an embedded operating system (Integrated ENS) [2], [26] or stand-alone libraries (Standalone ENS) [27], [28]. ENSs follow a layered software architecture, each layer implementing a specific protocol on the TCP/IP stack.

```
1  static void prvCheckOptions(...) {
2   const unsigned char *pucPtr = ... ;
3   const unsigned char *pucLast = pucPtr +
4   (((pxTCPHeader->ucTCPOffset >> 4) - 5) << 2);
5   while(pucPtr < pucLast){
6    ...
7    else if(( pucPtr[0]  == TCP_OPT_MSS) &&
8      (pucPtr[1] == TCP_OPT_MSS_LEN)) {
9     uxNewMSS = usChar2u16(pucPtr + 2);
10    if(pxSocket->u.xTCP.usInitMSS > uxNewMSS){
11     ...
12     pxTCPWindow->xSize.
13     ulRxWindowLength = ((uint32_t) uxNewMSS) *
14     (pxTCPWindow->xSize.ulRxWindowLength /
15     ((uint32_t) uxNewMSS ));
16       ...
17   }}
18   pucPtr += ...
19   ...
20 }}
```

Listing 1: CVE-2018-16523 and CVE-2018-16524: Snippet showing a divide-by-zero defect triggered by the TCP MSS Option (green) and an out-of-bound read (blue) triggered by the TCP Data Offset. Both are in the FreeRTOS network stack. EmNetTest can recreate these vulnerabilities (Table VII).

ENSs vulnerabilities pose a greater threat than those of regular network stacks due to the absence of operating systems and hardware vulnerability protection mechanisms. Regular operating systems, *e.g.,* Linux, provide more protection in their OS design. This includes features such as Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), address space isolation, and Stack Canaries [29] that prevent the exploitation of memory vulnerabilities. Also, modern processors include no-execute (Nx) regions that prevent the unauthorized execution of codes in sensitive memory regions. Many embedded OSes and processors lack these features [3], [30], increasing the ease of vulnerability exploitation.

ENSs are designed for embedded systems, which are resource-constrained, have real-time requirements, and often lack common library support. ENSs are also tailored to the underlying embedded operating system's threading and scheduling semantics. Consequently, they differ from regular operating systems' network stacks, which use the POSIX standard [31] for portability. For example, the `accept` call in FreeRTOS blocks until a successful TCP connection is established or the timeout elapses. The `accept` call in lwIP is non-blocking and defines a callback that would be called on successful connection establishment. Meanwhile, Contiki-ng has no `accept` syscall. Instead, it uses an event-driven callback for all events, including a Socket connection event.

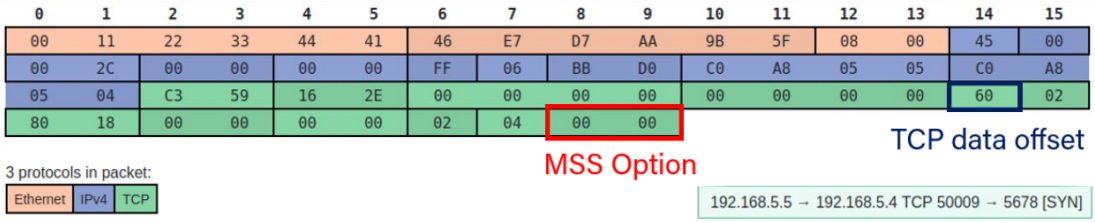

Fig. 1: A hex representation of a TCP packet showing the TCP header length field and the TCP MSS Option Value. Listing 1 describes two CVEs associated with these fields in the FreeRTOS ENS.

B. Internet Protocols and Network Packets

In this work, we focus on Internet Protocol (IP) or TCP/IP suite, which includes various protocols that specify how data should be packaged, addressed, and routed [32], [33]. The TCP/IP suite is organized into a layered architecture (Figure 2). An Internet packet has elements for each layer of this architecture, recursively structured as headers associated with one layer and a payload associated with the next (Figure 1). The protocol's implementation processes the corresponding headers at each level and passes the payload along.

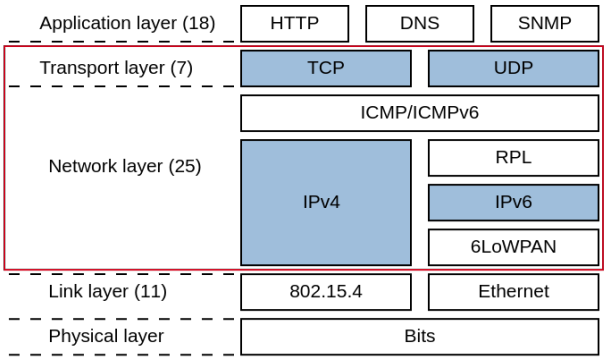

Fig. 2: Layers of the TCP/IP stack. Our tool targets layers in the red box (specific protocols in blue). Values in parentheses indicate number of analyzed CVEs in each layer (§V-B3).

C. ENS Vulnerabilities

ENSs are usually implemented in C/C++ for performance and compatibility reasons. They handle complex packet structures across multiple layers. Hence, ENSs are prone to defects that can be cybersecurity vulnerabilities. With the absence of sufficient protection mechanisms, the vulnerabilities in an ENS can be exploited to either disable or remotely control the entire system. Furthermore, these vulnerabilities can be triggered remotely by any use with network access to the system.

Listing 1 shows the snippet corresponding to two vulnerabilities, CVE-2018-16523 and CVE-2018-16524. CVE-2018-16523 (divide by zero) occurs because the TCP MSS option value, `uxNewMSS`, is used as the divisor to calculate `RxWindowLength`. A TCP packet with an MSS value of zero will lead to a divide-by-zero error. CVE-2018-16524 (out-of-bounds read) occurs because offset in the header, *i.e.,* `ucTCPOffset` is used to compute a pointer address `pucLast`, which is later read through `pucPtr`. These vulnerabilities are triggerable remotely without authorization by sending TCP packets. Furthermore, these vulnerabilities can be exploited to gain control of the system because of the lack of isolation mechanisms in embedded systems.

III. Related Work

a) Traditional Testing: Many ENSs incorporate test suites that help the maintainers validate the various functionalities they develop. As shown in §V, these test suites are inadequate. Although automated test generation tools [34]–[36] exist, the tests generated by them are inadequate at finding faults [37]. Furthermore, domain knowledge is required to use these automated test-generation tools effectively.

Research and commercial tools exist to facilitate the easy development of test suites for network stacks. PacketDrill [38], a network stack testing tool that enables the use of scripts to test the end-to-end correctness behavior of network stacks. PacketDrill focuses on testing functionality and always generates valid packets, *i.e.,* has valid and well-formed headers. However, as found in §V, most vulnerabilities occur because of invalid values in packet headers. InterWorking Labs has commercial testing solutions for testing network protocols and also uses malformed packets [39]. However, access costs over $10,000,[1] limiting its adoption in open-source projects and the low-margin embedded systems marketplace [40].

b) Formal Methods: Several tools [10]–[12] have explored formal methods for verifying network functions. Zaostrovnykh [10] and Pirelli [12] developed formal verification tools to automatically prove that a network function conforms to a provided specification. However, these techniques do not apply to multithreaded programs such as ENS. Microsoft's Project Everest [41] verifies various components of HTTPs and has provided verified implementations of some cryptographic libraries. FreeRTOS, maintained by AWS, also verifies their network stack implementation, FreeRTOS+TCP [42]. These formal methods are used to verify specific correctness properties of the network protocol implementations and do not make complete guarantees about their security. As shown by Fonseca *et al.* [43], formal methods guarantees are only as

[1]This quote was provided to us through personal communication.

good as their underlying assumptions. Hence, we still need to assess the security of these systems through dynamic testing.

c) Fuzzing: Fuzzing [44] has found many software defects. From a network perspective, fuzzing has been mainly explored to find bugs in network applications. AFLNet [18], StateAFL [19], and SnapFuzz [20] are three recent works in this direction. These works focus on setting up a proper communication channel with a network application and sending test data to the application through well-formed network packets. A recent work, TCPFuzz [21] uses fuzzing and differential testing to detect semantic vulnerabilities in the transport layer. TCPFuzz always generates valid packets and cannot find vulnerabilities arising from invalid packets.

Poncelet *et al.* [22] applied several state-of-the-art fuzzing tools to test individual functions of the Contiki-ng ENS. They reported that testing lower-layer functions does not get deep penetration. Conversely, directly testing upper network layers increases the rate of false positives as some inputs and corresponding packets are impossible as lower layers will reject them. Furthermore, they fail to trigger code paths that require the network stack to be in a particular state. This is demonstrated in our evaluation (§VII) where EmNetTest found various vulnerabilities in the well-test portions of Contiki-ng.

d) Vulnerability Studies: Several researchers have studied vulnerabilities' characteristics in different software systems [45]–[49]. Most of these works focus on well-provisioned systems, *e.g.,* desktop and web software. Few works study vulnerabilities in embedded systems. Al-Boghdady *et al.* [48] studied the characteristics of security vulnerabilities in IoT operating systems. While they focused on characterizing the CWEs (Common Weakness Enumeration) reported by static analysis tools, their study does not cover how these vulnerabilities are triggered or detected. Similar to our work, Malik & Pastore examined CVEs in Edge frameworks and found that (1) the network components were a common source of CVEs, and (2) specific values were often problematic, but did not go into detail on ENSs nor evaluate a solution [50]. Other industry practitioners have also published reports of security vulnerability analyses of ENSs they conducted. For example, Zimperium [51] published a blog post containing details of the vulnerabilities they discovered in FreeRTOS, and Forescout published a report containing an analysis of the 33 vulnerabilities they found and a list of observed common anti-patterns [5]. No prior work systematically analyzes vulnerabilities in ENSs.

IV. Knowledge Gaps and Research Questions

This work aims to fill two gaps. First, no study characterizes cybersecurity vulnerabilities in ENSs. Second, no dynamic system exists to systematically detect ENS cybersecurity vulnerabilities. We ask:

Theme 1: Vulnerability analysis

RQ1: What are the types and root causes of vulnerabilities?

RQ2: What packet sequences trigger ENS vulnerabilities?

Theme 2: State of practice for packet validation testing

RQ3: Are ENS tested for packet validation vulnerabilities?

Theme 3: Evaluating systematic testing with EmNetTest

RQ4: To what extent can bounded systematic testing uncover packet validation vulnerabilities?

V. ENS Vulnerabilities and Testing (RQ1-3)

This section presents methodology and results for RQ1-3. To summarize our findings, ENS CVEs are typically *packet validation vulnerabilities*. The studied ENS incorrectly handle packets that are slightly malformed, sometimes from a particular protocol state. In 95% of CVEs, 1-2 fields are incorrect. Repairs often involve a single `if`-statement.

A. Methodology

1) Repository Selection: We studied both integrated ENS and standalone ENS (Table I). We selected ENSs integrated into major open-source embedded operating systems. From lists in survey papers [53], [54], we selected three embedded OSes with over 1K GitHub stars: Zephyr (maintained by Linux Foundation), Contiki-ng (Supported by Swedish Research Institute), and FreeRTOS (maintained by AWS). From a previous vulnerability study [6], we selected the top-3 actively-maintained repositories (by GitHub stars) with reported CVEs. These were PicoTCP, LwIP, and FNet.

2) Data Collection: We obtained vulnerability reports (CVEs) from the National Vulnerability Database (NVD) [55]. We searched the NVD for the associated project. For integrated ENSs, we only considered vulnerabilities in the networking stack. There were 81 total CVEs. We discarded 15 CVEs that omitted technical vulnerability details. After preliminary analysis, we observed 61 of the remaining 66 vulnerabilities were caused by the poor validation of packets received by ENS. We termed these *packet validation (PV) vulnerabilities*. We removed the 5 non-PV vulnerabilities.

3) Data Analysis: One author analyzed each vulnerability report and technical details, including screenshots explaining vulnerable code, links to the vulnerability's GitHub issue, and the repairing pull request (PR).[2] We indicate the specific extracted features below — these are a typical set of features in software failure analysis [56]. For soundness, a second author analyzed a random sample of 13 vulnerabilities. We measured interrater agreement using Cohen's Kappa score [57]. We obtained κ=0.82, indicating substantial agreement [58].

B. RQ1: Vulnerability Characteristics

Finding 1: Memory Out-of-Bound Read and Write are the most common vulnerabilities (70%).
Finding 2: Missing length field validation and Missing packet size validation are the most frequent root causes and account for 69% of vulnerabilities.
Finding 3: The network layer contains most vulnerabilities (41%), followed by the application layer (29%). Vulnerabilities are also found in every layer of the stack.

We describe CVE types, root causes, and affected components.

[2]During this analysis, we found 3 new CVEs (excluded from our analysis).

Table I: Embedded network stacks (integrated and standalone) whose CVEs we examined. C/C++ LoC (source, not tests) measured with cloc [52]; for integrated ENS we measured only the network implementation. GitHub data as of May 2023.

Name	Size (LOC)	GitHub stars	GitHub forks	# CVEs studied	# CVEs recreated	# new vulns.
FreeRTOS(+TCP)	42.2K	3.6K (76)	1.2k (110)	11	5/5	0
Contiki-ng	41.6K	1.1K	635	24	2/2	2
Zephyr	95.7K	7.7K	4.8K	11	*Not attempted*	*Not attempted*
PicoTCP	32.7K	1K	201	12	5/7	4
LwIP	84.3K	525	249	1	*Not attempted*	1
FNET	18.0K	106	46	2	*Not attempted*	*Not attempted*

Table II: Proportion of CVE types. "Others": double-free, DNS cache poisoning, division-by-zero, and infinite loops.

Type	# CVEs (%)
Out-of-Bounds Read (CWE 125,126,200)	22 (36%)
Out-of-Bounds Write (CWE 120,121,122,787)	21 (34%)
Integer Overflow (CWE 191)	5 (8%)
Integer Underflow (CWE 190)	4 (7%)
Null-pointer dereference (CWE 476)	4 (7%)
Other	5 (8%)
Total	61 (100%)

Table III: Implementation-level root causes of CVEs.

Root cause	# CVEs (%)
Missing length field validation	23 (38%
Missing packet size validation	19 (31%)
Missing header value validation	7 (12%)
Missing integer wraparound validation	2 (3%)
Other	10 (16%)
Total	61 (100%)

1) Vulnerability Types: First, we group CVEs according to their Common Weakness Enumeration (CWE) [59]. Table II shows the result by this taxonomy. Memory over-read/write (the first two rows) comprise 70% of the vulnerabilities.

2) Implementation Root Causes: We studied code and repairs to learn the implementation-level root causes of CVEs. Table III groups these into several recurring patterns. Roughly 69% of CVEs in ENSs (first two rows) result from missing checks on length fields and data packet size. Two examples:

- **Missing length field validation (CVE-2018-16524):** FreeRTOS uses the TCP header length field to calculate the size of the TCP options region. However, it fails to validate the length value. Consequently, an invalid length value results in arbitrary memory read.
- **Missing packet size validation (CVE-2022-36054):** Contiki-ng receives a 6LoWPAN packet and after header compression, copies the packet into a buffer. If the packet is the first fragment of a fragmented packet, only 148 bytes are allocated. Contiki-ng doesn't verify the received packet size before copying it into this buffer. Consequently, a buffer overflow could result in a remote code execution attack.

3) Vulnerable Layers: The left column of Figure 2 shows the distribution of CVEs across the ENS layers. The top layers for CVEs are network (41%) and application (29%).

C. RQ2: Packet Sequences That Trigger CVEs

Finding 4: 95% of vulnerabilities depend on one or two fields and consequently can be triggered with a maximum of two field changes. 40 different fields contribute to these vulnerabilities.
Finding 5: 30% of CVEs are stateful, *e.g.,* involving an existing connection or a specific protocol state.

Here, we study packet sequences that can trigger these CVEs. Each packet sequence has a prefix (*i.e.,* state prefix) $p_1 p_2 \dots p_{k-1}$ that brings the ENS to a vulnerable state, followed by the vulnerability-triggering packet p_k. For instance, consider a vulnerability in processing a TCP FIN packet. To trigger the vulnerability, we first need to send packets that can set the ENS to a state where it accepts a FIN packet. Then, we send a FIN packet triggering the vulnerability. Understanding both parts enables a testing scheme to uncover real CVEs.

1) Properties of the vulnerability-triggering packets (p_k): Here, we investigate two aspects: (1) *Root Cause Fields* (RC_f): Which incorrectly-handled fields result in vulnerabilities? (2) *Dependent Fields* (D_f): How many fields of a packet does a vulnerability depend on?

For instance, consider CVE-2018-16599, which is caused by the incorrect validation of the UDP header length field. The vulnerability can be triggered only if the NBNS Type field is NET_BIOS (0x0020) and the NBNS Flags field indicates a response packet (0x8000). Here, RC_f = 1 (for the length field), whereas D_f = 3 (for the length, type, and flags fields).

Table IV shows RC_f and the number of CVEs resulting from it. We see that 57 (93%) of CVEs arise from fields in the protocol headers and options that are incorrectly handled. Table V shows the distribution of CVEs according to D_f. Most vulnerabilities (58, or 95%) have $D_f \leq 2$. However, it is not just one field that is problematic — 40 different fields across 15 protocols contribute to the 61 CVEs.

2) Properties of the packet sequence prefix: We studied the vulnerable code and execution path to identify any states involved. Table VI shows that 70% CVEs are stateless (can be triggered with a single packet/no prefix) and that the remaining

Table IV: Distribution of CVEs based on the incorrect fields (RC_f) in the CVE-triggering packet. These fields often included those specifying the length of the packet or option component (rows 1-2), or specific values of other fields or options (rows 3-4). Often, the packet was truncated (row 5).

Type	Count(%)
Header length value	8 (13%)
Option length value	8 (13%)
Header field value	24 (39%)
Option value	2 (3%)
Truncated packet	15 (25%)
Others	4 (7%)
Total	61 (100%)

Table V: Distribution of CVEs by # of dependent fields (D_f).

# Dependent Fields	# CVEs
1	34 (56%)
2	24 (39%)
> 2	3 (5%)
Total	61 (100%)

30% (12 CVEs) depend on the state of the system. Of these, 13 CVEs, occurring on stateful protocols, require the protocol to be in a specific set of states. 5 other CVEs depend on the properties of the previously-sent packet(s).

D. RQ3: Testing Suite Characteristic

We analyzed the test suites of four ENSs to understand why the known CVEs, which we discussed in §V-B, existed. Based on the CVE characteristics, we looked for four aspects of validation: (1) Unit tests involving input packet processing operations with malformed input; (2) Capability of injecting specific (and possibly malformed) packets; (3) Tests involving packets with invalid headers (cf. Table IV); and (4) Tests involving statefulness (cf. Table VI).

Table VI: Distribution of vulnerabilities based on the statefulness required to expose the vulnerability.

State Required	Protocol	# CVEs
Stateless	–	43 (70%)
Requires protocol state	TCP	6 (10%)
Requires protocol state	RPL	1 (2%
Requires protocol state	BLE	2 (3%)
Requires protocol state	MQTT	4 (7%)
Requires packet sequence	6LoWPAN	3 (5%)
Requires packet sequence	802.15.4	2 (3%)
Total	All	61 (100%)

Finding 6: ENSs are validated using end-to-end simulation tests and unit tests. The actual implementations of these tests are unique in each ENS (no standard test framework).
Finding 7: While some ENSs include packet injection tests, these are regression tests for specific CVEs. One ENS provides packet seeds and a harness for stateful fuzzing.
Finding 8: None of the ENSs systematically check invalid header or option fields, nor include unit tests for the various operations performed on an input packet.

FreeRTOS: FreeRTOS validates its ENS with end-to-end tests, unit tests, and formal verification. The end-to-end tests use the sockets interface to establish network connections and validate behaviors of the network stack. They provide (incomplete) memory safety proofs for main packet processing functions. Not all functions are verified and the provided proofs depend on the corrections of some unverified functions. FreeRTOS has some packet injection tests with invalid headers, but all cases are regressions for past CVEs.

Contiki-ng: Contiki-ng is validated with network simulation using cooja [60], a packet injection test, and fuzzing. The network simulation tests involve various end-to-end tests under different simulated network environments. Their packet injection tests use a fixed set of network packets. These are mostly regression tests to check for previous defects or vulnerabilities. Their packet injection framework is also used for fuzzing.

PicoTCP: PicoTCP validates with unit tests and end-to-end demo applications. The provided unit tests are mostly on non-packet related tasks, such as IP address-to-string conversion and socket tests. The demo applications test supported protocols in different network environments.

LwIP: LwIP validates with unit tests, network stress testing, and fuzzing. Their unit tests mostly test the output operations of the network stack, not the input packet processing functions. Several unit tests configure the test socket to specific protocol states. For stress, they measure the reliability of simulated networks while increasing the number of nodes and messages in the network. They also provide a fuzzing harness and fuzzing seeds for the different protocols.

VI. EmNetTest: Design and Implementation

A. Design Requirements

Based on our findings from Theme 1, an automated testing framework to detect PV vulnerabilities in ENSs should have three characteristics:

- **Ability to Detect Memory Issues:** Based on Finding 1, it should detect memory corruption vulnerabilities.
- **Systematic Packet Generation:** Based on Findings 2 and 4, it should systematically generate valid test packets with incorrect header values and truncated headers.
- **Stateful:** Based on Findings 3 and 5, it should drive the ENS stack to different protocol states for multiple protocols.

Such a framework would improve the state of the art in ENS testing (cf. §III and Findings 6-8).

B. Design

EmNetTest meets these requirements. Figure 3 illustrates.

- **Memory focused:** Address Sanitization (ASAN) [61] with ENS specific instrumentation.
- **Systematic packet generation:** An ordered generation algorithm can systematically generate all packets but is prioritized for packets that trigger known PV CVEs.
- **Stateful:** We build on the PACKETDRILL tool [38]. It provides packet sequences that cover some relevant protocols. We extend it to additional protocols and employ a seed set of state-covering sequences (packet sequence prefixes).

```
1  # Input: Num. entities N, stride S, packet p_k
2  # Output: Yields next packet for this config.
3
4  # SELECTION of N fields and options
5  {f_1,⋯,o_1,⋯} = nextPacketEntities(N) # Generator
6
7  # INTERPOLATION
8  vals_{f_1} = interpolate(f_1, S)
9  ...
10 vals_{o_1} = interpolate(o_1, S)
11 ...
12
13 # GENERATION (uses Python itertools)
14 for f_1 in vals_{f_1}:
15   ...
16     for o_1 in vals_{o_1}:
17       ...
18         p_k.modify(f_1, ⋯, o_i, ⋯)
19           yield p_k
20 # Caller sends prefix + |p_k| and checks result
```

Listing 2: Systematic packet generation. The caller imposes order by working from smaller to larger N and varying S.

1) Custom Address Sanitization with Dynamic Address Poisoning (DAP): Memory corruptions (*i.e.,* out-of-bounds read and write) in embedded systems may not lead to program crashes [62] (`SEGSEGV`). ASAN is a well-known technique to convert memory corruptions into program crashes. However, ASAN assumes the target application is using standard memory allocation and deallocation functions (*e.g.,* `malloc`/`free`) — which is not the case with ENSs, as they use custom allocators. To handle this, we use ASAN's Dynamic Address Poisoning (DAP) support. For each ENS, we modified its custom allocators such that after every allocation, the corresponding memory chunk will be unpoisoned (*i.e.,* OK to use). Similarly, we modify deallocator or release functions to poison the corresponding memory chunk (*i.e.,* Invalid to use).

We modify packer copying routines to detect out-of-bound memory accesses during packet processing. Specifically, after a packet is received and copied into the buffer, we poison the rest of the allocated buffer that is not covered by the received packet. Then, ASAN will detect any bytes read or written beyond the bounds of the allocated buffer.

2) (Ordered) Systematic Packet Generation: We systematically generate ordered test packets. *Systematic* means all packets are generated. *Ordered* means an ordering over the packets such that likely-useful packets are generated early.

We describe our approach, formalized in Listing 2. We assume a prefix sequence of packets $p_1 p_2 \dots p_{k-1}$ to reach a desired protocol state, followed by test packet p_k (§V-C). The algorithm generates all valuations of p_k.

Systematic: Packet p_k is a sequence of bytes consisting of required header fields, optional fields, and a payload. The payload was not the cause of ENS CVEs (§V) so we exclude it. Different subsets of the header and option fields are selected to modify (generator on line 5). We obtain possible values for each following an interpolation sequence from minimum (*e.g.,* `0x00`) to maximum (*e.g.,* `0xff`) along a stride S (line 7). All combinations are explored (line 13). This ensures we can detect vulnerabilities that either depend on the minima or maxima, or on a range of values as determined by the stride. To generate all p_k, choose maximum N and a Stride of 1.

Ordered: We order the generation of p_k starting from 0 entities (the original p_k), then 1 entity, and so on, discarding repeating packets. This order places the likely-to-be-useful packets early in the sequence — per Table V, most CVEs depend on at most 2 fields (*i.e.,* $D_f \leq 2$). This suggests that most of the vulnerabilities could be found with $N = 2$. During ENS validation, engineers may parameterize by bounding the maximum number of fields to select N. They may trade exhaustiveness vs. cost via the search stride S.

Handling truncate: As reported in Table IV, 25% of the analyzed CVEs involved a packet that was truncated to shorter than the expected length. The caller of Listing 2 generates these with modest post-processing: remove bytes from the end of the packet and then update checksums in earlier layers.

3) Stateful: In our CVE study, we found that many CVEs can only be triggered from certain states of a protocol. We examined existing network testing tools to identify one that can reach many states of a protocol. We chose the PACKETDRILL tool. It is designed to drive a network stack through the state machine for various protocols [38]. It supported two transport-layer protocols (TCP, UDP) and two network-layer protocols (IPv4, IPv6), with a corpus of >200 scripts that test different functionalities of the TCP protocol.

We developed a corpus of 7 Packetdrill scripts that can reach the 7 different TCP states where a packet can be injected (LISTEN, SYN-SENT, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, LAST-ACK, CLOSE-WAIT). We had only one UDP script as UDP has no states. We use these scripts as test script templates. As shown in Figure 3, for each test case and mutation instruction generated, we append the mutation instruction to all template scripts of the protocol being tested. This enables us to test all protocol states with the same input.

C. Implementation

Our EmNetTest implementation is 3,426 lines of C/C++ and Python. We describe pertinent aspects of the implementation.

1) Portability: The purpose of EmNetTest is to support many ENSs. Portability is a priority. As noted in §II, ENS have diverse architecture and semantics. We de-coupled the EmNetTest packet generation from the delivery and evaluation of packets. PacketDrill generates a combination of socket

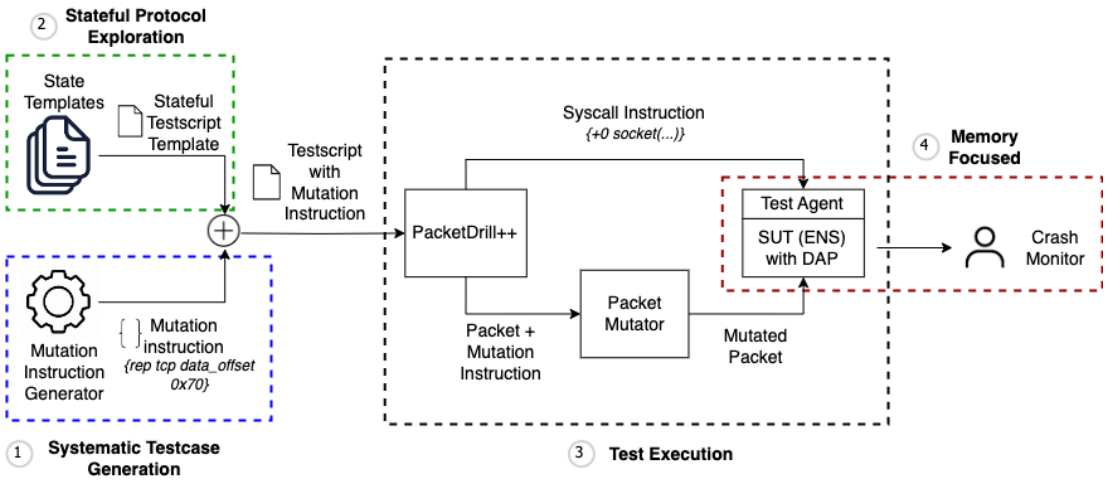

Fig. 3: Overview of the design of EmNetTest. It systematically generates mutation instructions by repeatedly taking a protocol header, selecting combinations of fields, and iterating through possible values (blue box ①). It achieves statefulness by using a set of test script templates that can explore different protocol states (green box ②). Packetdrill++ interprets each test script and sends the syscall and packets to the SUT (ENS). The per-ENS Test Agent maps received POSIX syscall instructions to appropriate behavior and execute the behavior on the ENS (black box ③). The SUT is instrumented with dynamic poisoning and compiled with ASAN to aid the detection of memory corruption (red box ④).

commands and network packets. As shown in Figure 3, a per-ENS Test Agent maps POSIX socket commands to the appropriate behavior on the ENSs. This includes both minor naming changes (*e.g.,* socket vs FreeRTOS_socket) as well as more substantial semantic changes (*e.g.,* rendering the asynchronous socket semantics of LwIP into synchronous POSIX semantics). This test agent (server) on the ENS receives socket interactions and packets and delivers them to the ENS. The rest of the system is agnostic to the ENS under test.

2) PACKETDRILL++*:* We implemented the network testing tool as an extension of PACKETDRILL. We extended the PACKETDRILL grammar to support mutation instructions. We implemented a packet mutator component in C that, given a packet and a set of mutation instructions, mutates the packet following the instructions. For example, the instructions might be to change the value of a field, insert an option, and truncate the packet. We modified PACKETDRILL so that it loads the Packet mutator as a shared library and uses it for packet mutation. PACKETDRILL++ also interacts with our portable bridge component instead of directly with the network.

3) Packet Injection: EmNetTest uses a virtual network interface (TAP [63]) to send mutated packets to Embedded Network Stack. A virtual network interface allows us to inject packets at the lowest layer of the network stack, which simulates the exact same behavior as when the ENS receives the packet from the internet, removing the possibility of false positives. Furthermore, a virtual network interface does not introduce the same network latency that would be introduced by a normal network interface connected to the internet.

4) Parallelizing test execution: Once packets are generated systematically, executing them is an embarrassingly parallel problem. We decoupled test case generation from execution using the producer-consumer pattern, saturating our servers.

5) Deduplicating vulnerabilities: EmNetTest systematically generates packets, which may result in many redundant defects. Our crash monitor analyzes the observed failures and deduplicates them based on the stack trace (line of crash).

6) Linux versions of ENSs: Although ENSs support many boards, they also support Linux as a development environment [64]. EmNetTest uses the Linux versions of the ENSs. This enables EmNetTest and the ENS to run on the same machine, enhancing communication between them. This does introduce the risk that our results mask defects in HW/SW integration on real boards, *e.g.,* due to layering issues [65].

VII. RQ4: SYSTEMATIC TESTING WITH EMNETTEST

We evaluate our systematic testing framework by running EmNetTest on 4 embedded network stacks. Our evaluation aims to understand the extent our systematic testing approach can uncover packet validation vulnerabilities. Specifically, we answer the following questions.

- **RQ4.1:** Can EmNetTest replicate known vulnerabilities?
- **RQ4.2:** Can EmNetTest discover new vulnerabilities?
- **RQ4.3:** What are EmNetTest's performance characteristics?
- **RQ4.4:** How does EmNetTest compare to fuzzing?

1) Experimental Setup: We evaluated N = 1,2,3 and we used a stride that yielded 4-6 values for each field (Listing 2).

We used the following servers for our experiments: two 32-core machines (Ubuntu 22.04, Intel Xeon W-2295 CPU@3GHz); and one 64-core machine (Ubuntu 22.04, AMD EPYC 7543P CPU@2.8GHz).

A. Embedded Network Stack Selection

We selected 4 ENSs for our evaluation — FreeRTOS+TCP, Contiki-ng, PicoTCP, and LWIP. These stacks from §V had

the highest proportion of Network and Transport layer vulnerabilities, suiting them for EmNetTest.

B. `ENSBench`: Vulnerability Dataset Contruction

To enable us to answer RQ4.1, we replicated 12 known vulnerabilities in recent versions of 3 selected ENSs — FreeRTOS, Contiki-ng, and PicoTCP.[3] These were selected out of the 14 reported vulnerabilities that affected the IPv4, IPv6, TCP, and UDP protocols in the selected ENSs layer protocols. We skipped 2 vulnerabilities because Packetdrill lacked support for the features they required (IPv6 fragmentation). We studied their fixing commits to replicate the vulnerabilities and reverted the fix. Porting the vulnerabilities to the latest version allowed us to have all vulnerabilities in a single build for testing. Table VII describes the CVEs we recreated.

Table VII: CVEs EmNetTest recreates. The last column indicates dependent fields and kind of changes that expose CVE. Notation: F—set header **F**ield; O—insert+set **O**ption; T—Truncate header; Rd—**Rea**d; Wr—**Wr**ite.

ENS	CVE-ID	Type	Operators
FreeRTOS	2018-16523	Div-by-zero	1 (O)
	2018-16524	OOB Read	1 (F)
	2018-16526	OOB Write	1 (O)
	2018-16601	Integer underflow	1 (F)
	2018-16603	OOB Read	1 (T)
Contiki-ng	2021-21281	OOB Read	1 (F)
	2022-36053	OOB Write	2 (F, T)
PicoTCP	2020-17441	OOB Read	2 (F, F)
	2020-17442	Integer Overflow	2 (F, O)
	2020-17444	Integer Overflow	2 (F, O)
	2020-17445	OOB Read	2 (F, O)
	2020-24337	Infinite Loop	1 (O)

C. RQ4.1: Replicating Known Vulnerabilities

To evaluate EmNetTest's ability to expose defects, we ran EmNetTest on vulnerable versions of FreeRTOS, Contiki-ng and PicoTCP. Our test found all vulnerabilities in the tested stacks as listed in Table VII using a maximum of 2 mutations. Table VII also shows the mutation types performed on the packet that exposed each vulnerability. These vulnerabilities were triggered by a total of 9 distinct fields. IPv6 extension header length caused 3 while TCP data offset caused 2.

D. RQ4.2: Discovering New Vulnerabilities

To evaluate EmNetTest's ability to discover new defects, we ran EmNetTest on recent versions of the ENS listed in §VII-B. For FreeRTOS and Contiki, we only ran the tests for IPv4 and IPv6 respectively as that was the only IP version they supported. Table I shows the count of vulnerabilities we found in each of the selected stacks. Table VIII describes the various vulnerabilities that we found. In our artifact, we also included the specific scripts that exposed each vulnerability and a detailed description and impact of each vulnerability.

Contiki-ng and PicoTCP confirmed the vulnerabilities we reported, assigned CVE identifiers, and repaired the vulnerabilities. We have been unable to establish communication with the LwIP team.

Table VIII: New vulnerabilities EmNetTest found. Notation: Same as Table VII.

ENS	CVE ID	Description	Config.
FreeRTOS	—	*No vuln found*	
Contiki-ng	2023-34100	OOB Rd (TCP MSS)	1 (O)
	2023-37459	OOB Rd (TCP flags)	1 (T)
PicoTCP	2023-35847	Div-by-zero (TCP MSS)	1 (O)
	2023-35846	OOB Rd (TCP fields)	1 (T)
	2023-35849	OOB Rd (IP checksum)	1 (F)
	2023-35848	OOB Rd (TCP MSS)	2 (O, O)
LwIP	L1	OOB Rd (TCP options)	1 (O)

We attempted to evaluate EmNetTest on commercial ENSs. We contacted five vendors of real-time OSes and embedded network stacks: WindRiver (VxWorks), Segger (emPower OS, embOS, emNet), Green Hills Software (GHNet), Lynx (LynxOS), and Sysgo (PikeOS). All declined to allow us to evaluate on their systems.

E. RQ4.3: Performance Characteristics

Test Execution Duration: We measured the execution duration of EmNetTest by varying the number of dependent fields (*i.e.,* N in Listing 2). Table IX shows the test execution duration on PicoTCP. For each value of N, we executed tests over all supported protocols (IPv4, IPv6, TCP, UDP) using 32 consumer instances. Our results in §VII-C and §VII-D show that all reported and new vulnerabilities could be found with only N=1 and N=2 tests.

Table IX: Performance results from testing on PicoTCP.

Task	# Test cases	Instances	Time
One test case	1	1	0.5 sec
N=1 test	2,211	32	2.13 min
N=2 test	134,296	32	2.21 hr
N=3 test	5,303,604	32	63.17 hr

Coverage Analysis: We analyzed the coverage achieved by running EmNetTest on 4 ENSs compiled with `gcov`.

Table X shows the line coverage achieved executing different tests. For the Integrated ENSs, we consider only the coverage of the networking component. The first two rows show the coverage for two PACKETDRILL tests with scripts representing different TCP states. The third row represents the coverage achieved when we ran stateful test scripts representing all TCP states. The last row indicates the coverage

[3]LwIP had no reported IP/TCP vulnerabilities.

Table X: Table showing the line coverage achieved by different tests when executing EmNetTest on all the tested stacks.

Test	FreeRTOS	Contiki	PicoTCP	lwIP
Script 1	37.4%	25.4%	11.3%	32.6%
Script 2	34.1%	24.8%	5.9%	29.9%
All Scripts	51.3%	29.6%	12.7%	40.0%
N=1 tests	53.4%	33.4%	14.8%	43.6%

when we ran a systematic test with N=1. By using test scripts that represent different TCP states, we achieved a significant increase in coverage. The little coverage increase caused by N=1 tests shows that packet validation vulnerabilities exist in codes that are covered by normal executions. As shown in Table VIII, this N=1 was also sufficient in detecting most of the new vulnerabilities we found. We could not get very high coverage as the ENSs contained protocol implementations in other network layers that we don't currently support.

F. RQ4.4: Fuzzing Comparison

We used the Contiki-ng fuzzing benchmark provided by Poncelet *et al.* [66] to demonstrate that within a time budget, fuzzing is not deterministic in uncovering vulnerabilities. We selected 4 fuzzers from the benchmark (MOpt [67], Intriguer [68], SymCC [69], and AFL). The first 3 had the best results during Poncelet *et al.*'s evaluations. AFL is a standard comparison point. To help the fuzzers, we (1) disabled checksums in Contiki-ng, and (2) augmented the fuzzers' seed set with EmNetTest's comprehensive set of seed packets.

Table XI: Fuzzing results with the Contiki-ng fuzzing benchmarks after 24 hours. Version 1 contains a version of Contiki-ng used by the Poncelet *et al.* authors for evaluation. It contains the vulnerabilities reported by the authors in their paper. Version 2 is the most recent commit on Contiki-ng on GitHub as of May 1st, 2023. This version contains 5 vulnerabilities, including 2 detected by EmNetTest. These vulnerabilities should cause a crash in V2 if triggered.

Metrics	Version 1 [22]	Version 2
# paths covered	190	316
Crashes found (#)	21	0
Hangs found (#)	12	0

After 24 hours, the second column of Table XI shows none of the fuzzers triggered any crash in vulnerable Version 2.

Comparison with other network protocol fuzzers: As noted in §III, there are other network fuzzers, *e.g.,* TCPFuzz [21] and AFLNet [18]. They are not appropriate for the vulnerabilities we studied. For example, AFLNet targets the application layer of the network stack, while TCPFuzz is concerned with semantic defects on legitimate input.

VIII. Discussion

EmNetTest vs Fuzzing vs Static Analysis: Fuzzing is the most used technique for detecting vulnerabilities. As a dynamic analysis technique, fuzzing provides a low rate of false positives. But fuzzing requires significant computing resources to be effective, limiting developers' ability to detect vulnerabilities at development time.

Like fuzzing, EmNetTest is also a dynamic analysis technique. But unlike fuzzing, it completes and provides guarantees that the known patterns of packet validation vulnerabilities do not exist in the ENS.

Static Analysis is another widely used approach that succeeds in detecting specific vulnerability patterns. Unlike dynamic analysis, many static analysis techniques consider only specific code sections rather than the entire software and give off a lot of false positives [70]. We briefly ran CodeQL [71] on the ENSs repositories and found that it struggled with inter-procedural cases, failing to find any known or new vulnerabilities we detected.

Learning from Intra and Inter-product Vulnerabilities: Our results in §VII-D show that the known vulnerability patterns still exist in ENSs. We found cases in Contiki-ng where they added regression tests for individual errors but failed to generalize these tests to classes of errors. We recommend that software engineers learn from the individual errors that occur in their software, and prepare generalized test cases that can detect similar errors.

We also found that the same vulnerabilities recur in different software implementations. For example, CVE-2023-35847 (§VII-D) in PicoTCP is the same as CVE-2018-16523 in FreeRTOS. CVE-2023-34100 in Contiki-ng is the same as CVE-2018-16524 in FreeRTOS. Anandayuvaraj *et al.* already performed preliminary studies on this phenomenon of recurring failures in software engineering [72] and initiated conversations towards a failure-aware software development lifecycle [23], [73]. Our findings in this paper further emphasize this need for software engineers to learn from the reported vulnerabilities and failures of other software products. Furthermore, tools like EmNetTest can help by ensuring that new vulnerability patterns, discovered in one software product, can be easily detected and fixed in every other similar software product they may exist in.

Integrating Security Protections to Embedded Firmware: As shown in §V-B, memory corruption is the most prevalent class of vulnerability reported in ENSs. In addition to detecting and fixing these vulnerabilities, protection mechanisms could also be implemented to harden the embedded devices and mitigate the impact of exploitations. Protection techniques such as stack canaries, Address Space Layout Randomization (ASLR), and No-Execute (Nx) regions exist for regular operating systems which makes vulnerability exploitation difficult. Unfortunately, Yu *et al.* [3] showed that these security protection techniques are missing in embedded systems. Prior research [40], [74], [75] identified cost as a factor that limits the integration of security in embedded systems. Our work

further illustrates the importance of integrating these security defenses into embedded systems. Hence, we advocate for further research in developing and deploying cost-effective protection mechanisms in embedded systems.

Improving Testing Practices for Open-source Software: As seen in §V-D, different ENSs employ diverse methods and implementations for testing. While many of the test suites had unit tests, the size and robustness of the tests varied across different ENSs. This suggests the need for a standard test framework for testing similar software systems such as ENSs.

Future Works: We identify the following opportunities for research to improve this work.

- *Checkpoint-based Optimization:* Stateful testing involves driving the ENS to a specific state before injecting the test packet. This introduces significant overhead. The use of a deferred forkserver [76] does not work on multi-threaded or networked applications. In the future, we hope to explore the use of process checkpointing and recovery [77] to optimize the execution of stateful tests.
- *Smart Test Case Generation:* Our current EmNetTest design depends on generating packets where all combinations of fields, up to a value k, can be mutated at a time. We plan to explore using program and dataflow analysis to understand which packet fields interact or depend on each other during packet processing and prioritize testing the combinations of these interacting fields. This approach will build on existing research on concolic and hybrid testing that integrates static analysis, dynamic analysis, and symbolic execution to aid vulnerability detection [78], [79]. This improvement will lead to optimizations in the time to run multiple field combinations shown in §VII-E.
- *Application to Other Protocols:* Our results show that vulnerabilities in all network protocol implementations share similar patterns. Hence, we have two questions. Would we find similar patterns in the implementations of other protocols? Would vulnerabilities in different implementations of the same protocols or protocol groups (*e.g.,* cryptographic protocols) share the same patterns?

IX. Limitations and Threats to Validity

Limitations of EmNetTest: While EmNetTest is effective in discovering packet validation vulnerabilities, it has the following limitations

- Our implementation of EmNetTest is limited to the protocols supported by Packetdrill (TCP, UDP, IPv4, IPv6, and ICMP). We believe EmNetTest will work for protocols in other layers as they contain vulnerabilities with similar patterns.
- Due to the different architectures of ENSs, EmNetTest requires a distinct Test Agent for each ENSs. We designed a portability layer that makes it easy to implement a Test Agent for any ENS.

Construct Validity: We studied CVEs using well-known classifications, minimizing construct-related risks. To mitigate further, we used inter-rater agreement as a check.

Internal Validity: We assessed the testing practices of ENSs by looking at their test suites. The maintainers of these ENSs may have other testing processes which we don't know about.

External Validity: We mitigate one generalizability concern by examining multiple ENS of both kinds (integrated and standalone). We acknowledge that the CVEs in our study (§V) may have been found by a small number of persons using specific techniques. There may be other vulnerability patterns in ENSs not detected or reported. Nevertheless, the patterns we observed in these CVEs helped us find new vulnerabilities.

X. Conclusion

Embedded Network Stacks play an important role in enabling interconnectivity in cyber-physical systems. Vulnerabilities in these stacks can have severe consequences. We conducted the first study of packet validation vulnerabilities in ENS. We studied 61 vulnerabilities in 6 ENSs. Our results revealed the root causes of packet validation vulnerabilities and the packet sequences needed to trigger them. We found that detecting many of these vulnerabilities required only simple mutations to the test packet. We designed EmNetTest following these findings and evaluated our implementation on 4 ENSs. We discovered 12 known and 7 new vulnerabilities. Our results show that appropriate systematic testing techniques can aid the timely and guaranteed detection of specific vulnerability classes. Other non-deterministic dynamic analysis techniques, such as fuzzing, should be deferred until applications have been adequately tested. Furthermore, EmNetTest will help maintainers of ENSs detect these packet validation vulnerabilities before deploying to the public.

XI. Data Availability

Our artifact is available at https://doi.org/10.5281/zenodo.8247917. In it, we provide:

1) A spreadsheet containing our analysis of known ENSs vulnerabilities.
2) The source code of EmNetTest and subcomponents.
3) ENSBench: Dataset of PacketDrill scripts to trigger the known and new vulnerabilities reported in this paper.

XII. Acknowledgements

We are grateful to the reviewers as well as J. Jones, W. Davis, and S. Bagchi for feedback on the manuscript. We thank P. Doshi and K. Robinson for assistance in data collection. J. Davis acknowledges support from NSF #2135156. A. Machiry acknowledges support from NSF #2247686 and DARPA N6600120C4031. Davis and Machiry acknowledge support from Rolls Royce.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or the United States Government.

REFERENCES

[1] A. Stanoev, "Contiki-NG: The OS for Next Generation IoT Devices." [Online]. Available: https://github.com/contiki-ng/contiki-ng/wiki/Home

[2] FreeRTOS, "FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions." [Online]. Available: https://www.freertos.org/index.html

[3] R. Yu, F. Del Nin, Y. Zhang, S. Huang, P. Kaliyar, S. Zakto, M. Conti, G. Portokalidis, and J. Xu, "Building Embedded Systems Like It's 1996," arXiv, Tech. Rep. arXiv:2203.06834, Mar. 2022, arXiv:2203.06834 [cs] type: article. [Online]. Available: http://arxiv.org/abs/2203.06834

[4] L. H. Newman, "An Operating System Bug Exposes 200 Million Critical Devices," *Wired*, section: tags. [Online]. Available: https://www.wired.com/story/vxworks-vulnerabilities-urgent11/

[5] "AMNESIA:33." [Online]. Available: https://www.forescout.com/research-labs/amnesia33/

[6] Forescout, "Project Memoria." [Online]. Available: https://www.forescout.com/research-labs/project-memoria/

[7] L. Lockefeer, D. M. Williams, and W. Fokkink, "Formal specification and verification of TCP extended with the Window Scale Option," *Science of Computer Programming*, vol. 118, pp. 3–23, Mar. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642315001835

[8] M. A. S. Smith, "Formal Verification of TCP and T/TCP."

[9] M. Musuvathi and D. R. Engler, "Model Checking Large Network Protocol Implementations."

[10] A. Zaostrovnykh, S. Pirelli, R. Iyer, M. Rizzo, L. Pedrosa, K. Argyraki, and G. Candea, "Verifying software network functions with no verification expertise," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 275–290. [Online]. Available: https://dl.acm.org/doi/10.1145/3341301.3359647

[11] K. Zhang, D. Zhuo, A. Akella, and A. K. X. Wang, "Automated Verification of Customizable Middlebox Properties with Gravel."

[12] S. Pirelli, A. Valentukonytė, K. Argyraki, and G. Candea, "Automated Verification of Network Function Binaries," 2022, pp. 585–600. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/pirelli

[13] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao, "Static Detection of Packet Injection Vulnerabilities: A Case for Identifying Attacker-controlled Implicit Information Leaks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 388–400. [Online]. Available: https://dl.acm.org/doi/10.1145/2810103.2813643

[14] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically Discovering High-Order Taint Style Vulnerabilities in OS Kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, Nov. 2021, pp. 811–824. [Online]. Available: https://dl.acm.org/doi/10.1145/3460120.3484798

[15] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," in *Proceedings of the ACM SIGCOMM 2011 conference*, ser. SIGCOMM '11. New York, NY, USA: Association for Computing Machinery, Aug. 2011, pp. 26–37. [Online]. Available: https://dl.acm.org/doi/10.1145/2018436.2018440

[16] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov, "Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities," in *2009 22nd IEEE Computer Security Foundations Symposium*, Jul. 2009, pp. 186–199, iSSN: 2377-5459.

[17] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein, "Analyzing protocol implementations for interoperability," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. USA: USENIX Association, May 2015, pp. 485–498.

[18] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A Greybox Fuzzer for Network Protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Oct. 2020, pp. 460–465, iSSN: 2159-4848.

[19] R. Natella, "StateAFL: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, Dec. 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10233-3

[20] A. Andronidis and C. Cadar, "SnapFuzz: An Efficient Fuzzing Framework for Network Applications," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 2022, pp. 340–351, arXiv:2201.04048 [cs]. [Online]. Available: http://arxiv.org/abs/2201.04048

[21] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "{TCP-Fuzz}: Detecting Memory and Semantic Bugs in {TCP} Stacks with Fuzzing," 2021, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/zou

[22] C. Poncelet, K. Sagonas, and N. Tsiftes, "So Many Fuzzers, So Little Time: Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack," 2022. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-61138

[23] D. Anandayuvaraj, P. Thulluri, J. Figueroa, H. Shandilya, and J. C. Davis, "Towards a failure-aware SDLC for internet of things," 2022. [Online]. Available: https://arxiv.org/abs/2206.13562

[24] D. A. Norman, "Commentary: Human error and the design of computer systems," *Communications of the ACM*, vol. 33, no. 1, pp. 4–7, 1990, publisher: Association for Computing Machinery, Inc.

[25] C. Johnson, "Software Support for Incident Reporting Systems in Safety-Critical Applications," in *Proceedings of the 19th International Conference on Computer Safety, Reliability and Security*, ser. SAFECOMP '00. Berlin, Heidelberg: Springer-Verlag, Oct. 2000, pp. 96–106.

[26] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, and N. Tsiftes, "The Contiki-NG open source operating system for next generation IoT devices," *SoftwareX*, vol. 18, Jun. 2022, publisher: Elsevier. [Online]. Available: https://www.softxjournal.com/article/S2352-7110(22)00062-0/fulltext

[27] lwIP, "lwIP: Overview." [Online]. Available: https://www.nongnu.org/lwip/2_1_x/index.html

[28] PicoTCP, "picotcp." [Online]. Available: http://picotcp.altran.be/

[29] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[30] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in Designing Exploit Mitigations for Deeply Embedded Systems," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, Jun. 2019, pp. 31–46.

[31] IEEE, "IEEE Standard for IEEE Information Technology - Portable Operating System Interface (POSIX(TM))," *IEEE Std 1003.1-2001 (Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992)*, pp. 1–3678, Dec. 2001, conference Name: IEEE Std 1003.1-2001 (Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992).

[32] Robert T. Braden, "Requirements for Internet Hosts - Application and Support," Internet Engineering Task Force, Request for Comments RFC 1123, Oct. 1989, num Pages: 98. [Online]. Available: https://datatracker.ietf.org/doc/rfc1123

[33] R. T. Braden, "Requirements for Internet Hosts - Communication Layers," Internet Engineering Task Force, Request for Comments RFC 1122, Oct. 1989, num Pages: 116. [Online]. Available: https://datatracker.ietf.org/doc/rfc1122

[34] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, Sep. 2011, pp. 416–419. [Online]. Available: https://dl.acm.org/doi/10.1145/2025113.2025179

[35] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 75–84, iSSN: 1558-1225.

[36] D. Beyer, "Advances in Automatic Software Testing: Test-Comp 2022," in *International Conference on Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer-Verlag, Apr. 2022, pp. 321–335. [Online]. Available: https://doi.org/10.1007/978-3-030-99429-7_18

[37] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli, "On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, May 2019, pp. 121–125, iSSN: 2574-3864.

[38] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H.-k. J. Chu, A. Terzis, and T. Herbert, "packetdrill: Scriptable Network Stack Testing, from Sockets to Packets," 2013, pp.

213–218. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/cardwell
[39] I. InterWorking Labs, "Network testing products," 2022. [Online]. Available: https://www.iwl.com/products
[40] N. K. Gopalakrishna, D. Anandayuvaraj, A. Detti, F. L. Bland, S. Rahaman, and J. C. Davis, ""If security is required": Engineering and Security Practices for Machine Learning-based IoT Devices," in *4th International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT)*, 2022, p. 8.
[41] Microsoft, "Project Everest." [Online]. Available: https://www.microsoft.com/en-us/research/project/project-everest-verified-secure-implementations-https-ecosystem/
[42] N. Chong and B. Jacobs, "Formally verifying freertos' interprocess communication mechanism," in *Embedded World Exhibition & Conference 2021*, 2021. [Online]. Available: https://www.amazon.science/publications/formally-verifying-freertos-interprocess-communication-mechanism
[43] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An Empirical Study on the Correctness of Formally Verified Distributed Systems," in *Proceedings of the Twelfth European Conference on Computer Systems*. Belgrade Serbia: ACM, Apr. 2017, pp. 328–343. [Online]. Available: https://dl.acm.org/doi/10.1145/3064176.3064183
[44] 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, pp. 2312–2331, 2019.
[45] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, "A Large-Scale Empirical Study on Vulnerability Distribution within Projects and the Lessons Learned," in *IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.
[46] M. Jimenez, M. Papadakis, and Y. L. Traon, "An Empirical Analysis of Vulnerabilities in OpenSSL and the Linux Kernel," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2016, pp. 105–112, iSSN: 1530-1362.
[47] M. Cai, H. Huang, and J. Huang, "Understanding Security Vulnerabilities in File Systems," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '19. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 8–15. [Online]. Available: https://doi.org/10.1145/3343737.3343753
[48] A. Al-Boghdady, K. Wassif, and M. El-Ramly, "The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT's Low-End Devices," *Sensors*, vol. 21, no. 7, p. 2329, Jan. 2021, number: 7 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1424-8220/21/7/2329
[49] J. McBride, B. Arief, and J. Hernandez-Castro, "Security Analysis of Contiki IoT Operating System," in *Proceedings of the 2018 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN ’18. USA: Junction Publishing, Feb. 2018, pp. 278–283.
[50] J. Malik and F. Pastore, "An empirical study of vulnerabilities in edge frameworks to support security testing improvement," *Empirical Software Engineering*, vol. 28, no. 4, p. 99, 2023.
[51] Zimperium Labs, "FreeRTOS TCP/IP Stack Vulnerabilities - The Details." [Online]. Available: https://www.zimperium.com/blog/freertos-tcpip-stack-vulnerabilities-details/
[52] A. Danial, "cloc: Count lines of code," https://github.com/AlDanial/cloc, 2021, accessed: May 3, 2023.
[53] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, Oct. 2016, conference Name: IEEE Internet of Things Journal.
[54] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, "Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10 375–10 383, Dec. 2019, conference Name: IEEE Internet of Things Journal.
[55] "National vulnerability database," https://nvd.nist.gov/, accessed: May 3, 2023.
[56] P. Amusuo, A. Sharma, S. R. Rao, A. Vincent, and J. C. Davis, "Reflections on software failure analysis," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering — Ideas, Visions, and Reflections track (ESEC/FSE-IVR)*, 2022.
[57] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.
[58] J. R. Landis and G. G. Koch, "An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers," *Biometrics. Journal of the International Biometric Society*, pp. 363–374, 1977, publisher: JSTOR.
[59] "Common weakness enumeration," https://cwe.mitre.org/, accessed: May 3, 2023.
[60] Contiki, "An introduction to cooja," https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja, 2021, accessed: May 3, 2023.
[61] K. Serebryany and D. Bruening, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Annual Technical Conference*. USENIX Association, 2012, pp. 309–318.
[62] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." in *NDSS*, 2018.
[63] "Universal TUN/TAP device driver — The Linux Kernel documentation." [Online]. Available: https://docs.kernel.org/networking/tuntap.html
[64] J. Srinivasan, S. R. Tanksalkar, P. C. Amusuo, J. C. Davis, and A. Machiry, "Towards rehosting embedded applications as linux applications," in *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023.
[65] M. Shen, J. C. Davis, and A. Machiry, "Towards automated identification of layering violations in embedded applications," in *2023 ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2023.
[66] C. Poncelet, K. Sagonas, and N. Tsiftes, "So Many Fuzzers, So Little Time: Experience from Evaluating Fuzzers on the Contiki-NG Network (Hay)Stack," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, Jan. 2023, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3556946
[67] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "Mopt: Optimized mutation scheduling for fuzzers." in *USENIX Security Symposium*, 2019, pp. 1949–1966.
[68] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 515–530.
[69] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 181–198.
[70] A. S. Ami, K. Moran, D. Poshyvanyk, and A. Nadkarni, ""False negative – that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing," Aug. 2023, arXiv:2307.16325 [cs]. [Online]. Available: http://arxiv.org/abs/2307.16325
[71] "CodeQL." [Online]. Available: https://codeql.github.com/
[72] D. Anandayuvaraj and J. C. Davis, "Reflecting on Recurring Failures in IoT Development," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, Jan. 2023, pp. 1–5. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3559545
[73] D. Anandayuvaraj, P. Thulluri, J. Figueroa, H. Shandilya, and J. C. Davis, "Incorporating Failure Knowledge into Design Decisions for IoT Systems: A Controlled Experiment on Novices," in *Software Engineering Research & Practices for the Internet of Things (SERP4IoT)*, 2023.
[74] C. Bodei, S. Chessa, and L. Galletta, "Measuring security in IoT communications," *Theoretical Computer Science*, vol. 764, pp. 100–124, Apr. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397518307205
[75] W. Toussaint and A. Y. Ding, "Machine Learning Systems in the IoT: Trustworthiness Trade-offs for Edge Intelligence," in *2020 IEEE Second International Conference on Cognitive Machine Intelligence (CogMI)*, Oct. 2020, pp. 177–184.
[76] "More about AFL — AFL 2.53b documentation." [Online]. Available: https://afl-1.readthedocs.io/en/latest/about_afl.html
[77] "CRIU." [Online]. Available: https://criu.org/Main_Page
[78] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf
[79] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM} : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," 2018,

pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun