TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing

Yong-Hao Zou and Jia-Ju Bai, *Tsinghua University;* Jielong Zhou, Jianfeng Tan, and Chenggang Qin, *Ant Group;* Shi-Min Hu, *Tsinghua University*

https://www.usenix.org/conference/atc21/presentation/zou

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing

Yong-Hao Zou, Jia-Ju Bai
Tsinghua University

Jielong Zhou, Jiangfeng Tan, Chenggang Qin
Ant Group

Shi-Min Hu
Tsinghua University

Abstract

TCP stacks provide reliable data transmission in network, and thus they should be correctly implemented and well tested to ensure reliability and security. However, testing TCP stacks is difficult. First, a TCP stack accepts packets and system calls that have dependencies between each other, and thus generating effective test cases is challenging. Second, a TCP stack has various complex state transitions, but existing testing approaches target covering states instead of covering state transitions, and thus their testing coverage is limited. Finally, our study of TCP stack commits shows that 87% of bug-fixing commits are related to semantic bugs (such as RFC violations), but existing bug sanitizers can detect only memory bugs not semantic bugs.

In this paper, we design a novel fuzzing framework named TCP-Fuzz, to effectively test TCP stacks and detect bugs. TCP-Fuzz consists of three key techniques: (1) a *dependency-based strategy* that considers dependencies between packets and system calls, to generate effective test cases; (2) a *transition-guided fuzzing approach* that uses a new coverage metric named *branch transition* as program feedback, to improve the coverage of state transitions; (3) a *differential checker* that compares the outputs of multiple TCP stacks for the same inputs, to detect semantic bugs. We have evaluated TCP-Fuzz on five widely-used TCP stacks (TLDK, F-Stack, mTCP, FreeBSD TCP and Linux TCP), and find 56 real bugs (including 8 memory bugs and 48 semantic bugs). 40 of these bugs have been confirmed by related developers.

1 Introduction

The TCP protocol is a transport-layer network protocol that receives system calls and packets to provide reliable data transmission. It carries over 85% of network traffic nowadays [44, 63]. In practice, the TCP protocol has different implementations, forming different TCP stacks. Each modern operating system (such as Linux and FreeBSD) has its own kernel-level TCP stack to provide fundamental network support for user-level applications. Besides, to achieve better performance and reduce impact on OS kernels, many user-level TCP stacks (such as mTCP [26], TLDK [59] and F-Stack [19]) have been developed and widely-used in telecom systems and network nodes, to transfer data without OS involvement.

Though TCP stacks are critical, correctly implementing them is difficult [4, 16], as a TCP stack has rich functionalities (such as reliable transmission and congestion control), complex state model and various kinds of possible exceptions to handle. Thus, developers may unintentionally make mistakes when implementing TCP stacks, introducing bugs that can cause serious problems. Memory bugs (such as null-pointer dereferences and use-after-free issues) are common in TCP stacks, and they can cause crashes, data corruption and so on. Moreover, according to our study of TCP stack commits, 87% of bug-fixing commits are related to *semantic bugs* (such as RFC violations), which are related to code logics and RFC documents, instead of memory accesses. For example, CVE-2019-11478 [15] reports that the TCP retransmission queue in the Linux TCP stack can be fragmented when handling certain TCP Selective Acknowledgment (SACK) sequences, and attackers can exploit this bug to cause a denial of service. Thus, it is important to test TCP stacks to detect bugs.

To detect bugs in TCP stacks, some approaches [34, 40, 41, 53] use model checking or formal verification to check the correctness of TCP implementation. But they require much manual effort and TCP-specific knowledge to provide a complete and correct TCP state model, and they are often time-consuming due to high complexity of TCP state transitions. To reduce manual effort and time usage, some approaches [9, 30] perform static analysis of TCP stack source code. But they often introduce false positives, due to lacking exact runtime information. To reduce false positives, some approaches [3–5, 66] analyze the runtime traces of TCP stacks to infer RFC violations. However, they require substantial and effective test cases to achieve high testing coverage.

To generate effective test cases, many recent approaches perform fuzz testing for the implementations of application-layer network protocols, such as DTLS/TLS [17, 52, 54, 60],

FTP [6, 21, 43] and Modbus [37, 38]. But these approaches are limited in testing TCP stacks for three critical reasons: (1) These approaches generate just packets as input test cases, without considering dependencies between inputs; but TCP stacks receive both system calls (syscalls) and packets, which have dependencies between each other. Thus, these approaches are limited in generating effective test cases for TCP stacks. (2) These approaches use code coverage as program feedback to cover different protocol states; but besides states, TCP stacks also have various state transitions that heavily affect TCP execution and can trigger semantic bugs. Thus, these approaches fail to cover many state transitions and thus may miss many real bugs. (3) Many of these approaches use common bug sanitizers (such as ASan [2] and MSan [39]) to detect memory bugs; but many bugs in TCP stacks are semantic bugs that are unrelated to memory accesses, and thus common bug sanitizers cannot find these semantic bugs.

In this paper, we propose a novel TCP-stack fuzzing framework named TCP-Fuzz, which consists of three key techniques. First, to generate effective test cases, TCP-Fuzz uses a *dependency-based strategy* that can generate the sequences of syscalls and packets by considering dependencies between them. Specifically, this strategy considers three kinds of dependencies to generate effective test cases, including syscall-syscall, packet-packet and syscall-packet dependencies. For example, a typical packet-packet dependency is that the sequence number of a new packet should be equal to the sum of the sequence number and data length of the previous packet. Second, to effectively cover state transitions, TCP-Fuzz uses a *transition-guided fuzzing approach* that exploits a new coverage metric named *branch transition* as program feedback to replace code coverage. Branch transition is represented as a vector that stores both branch coverage for the current input item (packet or syscall) and the change of branch coverage between the current and previous input items. In this way, branch transition can describe not only states but also state transitions of two adjacent input items. Finally, to detect semantic bugs, TCP-Fuzz uses a *differential checker* that compares the outputs of multiple TCP stacks for the same inputs. Indeed, different TCP stacks should obey many identical semantic rules (most of these rules are defined in RFC documents), and thus they should produce identical or similar outputs for the same inputs. Otherwise, these TCP stacks have implementation inconsistencies, indicating some of them possibly have semantic bugs. This checker is scalable and does not introduce runtime overhead for TCP stacks.

We have implemented TCP-Fuzz with Clang [33] and Packetdrill [8]. TCP-Fuzz can detect both memory bugs with existing bug sanitizers and semantic bugs with our differential checker. Overall, we make four main contributions:

- We study TCP stack commits, and find 87% of bug-fixing commits are related to semantic bugs, which cannot be found by existing bug sanitizers. We also reveal the limitations of existing protocol fuzzing in testing TCP stacks.
- To improve fuzzing in testing TCP stacks, we propose three key techniques: (1) a *dependency-based strategy* that considers dependencies between packets and system calls, to generate effective test cases; (2) a *transition-guided fuzzing approach* that uses a new coverage metric named *branch transition* as fuzzing feedback, to improve the coverage of state transitions; (3) a *differential checker* that compares the outputs of multiple TCP stacks for the same inputs, to detect semantic bugs.
- Based on the three key techniques, we design a novel fuzzing framework named TCP-Fuzz, to effectively test TCP stacks. To our knowledge, TCP-Fuzz is the first systematic TCP-stack fuzzing framework to detect both memory and semantic bugs.
- We evaluate TCP-Fuzz on five widely-used user-level and kernel-level TCP stacks (TLDK, F-Stack, mTCP, FreeBSD TCP and Linux TCP), and find 56 real bugs (including 8 memory bugs and 48 semantic bugs). 40 of these bugs have been confirmed by related developers, and 23 bugs have been fixed. Moreover, we also compare TCP-Fuzz to existing fuzzing approaches (AFL-like, Syzkaller-like, Boofuzz, Fuzzotron and AFLNet), and it finds many real bugs missed by these approaches.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 introduces our key techniques of fuzzing TCP stacks. Section 4 introduces TCP-Fuzz. Section 5 shows our evaluation and compares TCP-Fuzz to existing fuzzing tools. Section 6 makes a discussion about fuzzing TCP stacks. Section 7 presents related work, and Section 8 concludes this paper.

2 Background and Motivation

We first introduce TCP stacks, and then we motivate our work by studying TCP stack commits and revealing the limitations of existing protocol fuzzing in testing TCP stacks.

2.1 TCP Stack

The TCP protocol is a classical transport-layer protocol to provide reliable, ordered and error-checked delivery of byte streams via an IP network. In practice, the TCP protocol has different implementations, forming different TCP stacks. Besides classical kernel-level TCP stacks (such as Linux TCP and FreeBSD TCP), many new user-level TCP stacks (such mTCP, TLDK and F-Stack) have been developed and widely-used to achieve better performance. However, all these TCP stacks has three common features:

F1: Two-dimensional inputs with dependencies. As presented in Figure 1, a TCP stack receives both packets from network drivers and syscalls from applications as inputs, and it outputs the syscalls' results to applications and response packets to network drivers. TCP-related system calls are used

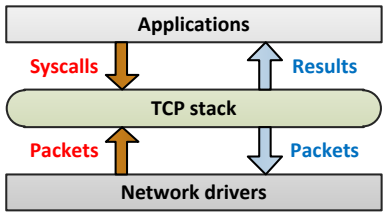

Figure 1: Inputs and outputs of TCP stack.

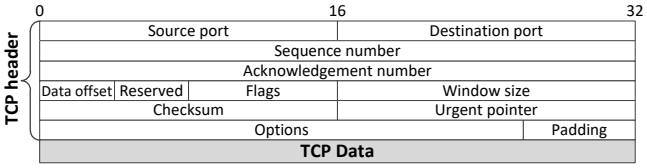

Figure 2: TCP packet format.

to perform fixed network functionalities. For example, the syscall `socket` is used to create an endpoint for communication and it returns a file descriptor of the socket; the syscall `accept` is used to accept a connection on a socket and it returns a new file descriptor of the socket. A TCP packet has a fixed format shown in Figure 2, including a header and data. The TCP header consists of different fields to store the parameters and state of an end-to-end TCP socket.

Packets and syscalls accepted by the TCP stack should have dependencies between each other, otherwise they will be simply neglected by the TCP stack without deep processing. Specifically, there are three kinds of dependencies:

- *Syscall-syscall dependency.* For example, when a connection is passive open, the application must call a series of syscalls including `socket`, `bind`, `listen` and `accept` in order. Otherwise, the application cannot successfully establish the TCP connection.
- *Packet-packet dependency.* For example, after a connection is established, the source port and destination port of each packet should be fixed. Otherwise, the TCP stack identifies the packets to be invalid and thus directly drops them without further processing.
- *Syscall-packet dependency.* For example, the syscall `accept` returns only after the TCP stack receives the last one of the three-way handshake packets.

According to this feature, two requirements should be satisfied when testing TCP stacks. First, it is necessary to generate the sequences of both system calls and packets as input test cases. Second, to make test cases more effective, it is important to consider dependencies between packets and syscalls when generating test cases.

F2: State model. A TCP stack works according to a basic state model defined in the RFC 793 [50] document. Figure 3 shows this basic state model, which has 11 states and 20 state transitions. For a real-world TCP stack, there are often more states and state transitions specific to the TCP stack's implementation.

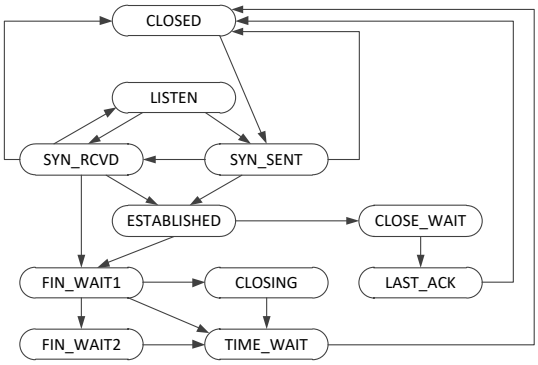

Figure 3: Basic state model of TCP stack.

According to this feature, when testing TCP stacks, it is important to cover both states and state transitions as many as possible. As a state can be reached from different states (for example in Figure 3, *TIME_WAIT* can be reached from *FIN_WAIT1*, *FIN_WAIT2* and *CLOSING*) and there are often more state transitions than states, covering state transitions is actually more important than covering states in testing.

F3: Semantic rules. Each TCP stack works based on some regular semantic rules that stipulate how syscalls and packets should be handled. Most of these semantic rules are explicitly described in RFC documents. For example, the RFC 7323 [49] document describes how to handle the timestamp option in the TCP packet header. An important semantic rule in this RFC document is that once the timestamp option has been successfully negotiated during TCP connection, the TCP stack should accept only packets with non-decreasing timestamps; otherwise the TCP stack should simply drop the packets. However, some semantic rules are not explicitly described in RFC documents. For example, RFC documents defines 32 possible options in the TCP packet header and describes how these options should be handled [58]. But for unknown options, RFC documents do not describe how to handle them. In practice, most TCP stacks simply ignore these options.

According to this feature, when testing TCP stacks, it is important to check these semantic rules and detect related violations. Indeed, these violations are unrelated to problematic memory accesses, and thus we refer them to *semantic bugs*.

2.2 Study of TCP Stack Commits

To understand the proportion of memory bugs and semantic bugs in existing TCP stacks, we select three open-sourced and widely-used TCP stacks, including FreeBSD TCP, mTCP [26] and TLDK [59], to study their commits[1]. Among these TCP stacks, FreeBSD TCP is a classical kernel-level TCP stack; mTCP is a well-known user-level TCP stack in academic community; TLDK is a recent user-level TCP stack in industry

[1] FreeBSD commits: https://gitlab.com/FreeBSD/freebsd-src
mTCP commits: https://github.com/mtcp-stack/mtcp
TLDK commits: https://git.fd.io/tldk/commit/?h=dev-next-socket

Time	FreeBSD		mTCP		TLDK	
	Memory	*Semantic*	*Memory*	*Semantic*	*Memory*	*Semantic*
2017	2	26	2	6	1	11
2018	9	51	0	4	0	4
2019	9	65	1	3	2	5
Total	20	142	3	13	3	20

Table 1: Study results of TCP stack commits.

community and it has been deployed in many telecom systems and network nodes. In our study, we first select the bug-fixing commits from January 2017 to December 2019, resulting in 201 commits; and then we manually read each commit to identify whether it fixes memory bugs or semantic bugs.

Table 1 shows the study results. 87% of bug-fixing commits are related to semantic bugs, namely most of the reported bugs in TCP stacks are semantic bugs. Figure 4 shows an example commit [14] of fixing a semantic bug in FreeBSD TCP stack. The annotation of this commit describes that it fixes a RFC 7323 [49] violation. Specifically, the TCP stack mistakenly accepted the packets with decreasing timestamp values. To fix the bug, this commit adds several checks about the timestamp value to drop invalid packets.

```
FILE: FreeBSD/sys/netinet/tcp_syncache.c
  int syncache_expand(...) {
        ......
+       /* RFC 7323 PAWS: If we have a timestamp on this segment and
+        * it is less than ts_recent, drop it.
+       if (sc->sc_flags & SCF_TIMESTAMP && to->to_flags & TOF_TS &&
+           TSTMP_LT(to->to_tsval, sc->sc_tsreflect)) {
+               SCH_UNLOCK(sch);
+               if ((s = tcp_log_addrs(inc, th, NULL, NULL))) {
+                       log(LOG_DEBUG, ...);
+                       free(s, M_TCPLOG);
+               }
+               return (-1);  /* Do not send RST */
+       }
        ......
  }
```

Figure 4: Example commit of fixing a semantic bug.

In fact, these semantic bugs are introduced for three main reasons. First, because a TCP stack has rich functionalities and complex state model, developers may unintentionally make mistakes about semantic rules when implemented the TCP stack. Second, many semantic rules are used to handle exceptions that infrequently occur in normal execution, and thus the code related to these rules receives insufficient attention in development and testing. Finally, some semantic rules are not explicitly described in RFC documents, and thus developers cannot ensure whether their implemented code obeys these rules. For these reasons, it is important to find semantic bugs in TCP stacks.

2.3 Limitations of Existing Protocol Fuzzing

Fuzzing is an effective technique of runtime testing, and it has shown excellent ability of bug detection in practice. Encouraged by the promising results, many recent approaches perform fuzz testing for the implementations of application-layer network protocols, such as DTLS/TLS [17, 52, 54, 60], FTP [6, 21, 22, 43] and Modbus [37, 38]. However, we believe that these approaches are limited in testing TCP stacks for three critical reasons:

1) Fail to generate two-dimensional inputs with dependencies. Existing fuzzing approaches only generate packets as input test cases, without considering dependencies between inputs. However, as described in *F1* in Section 2.1, TCP stacks receive both syscalls and packets, which have dependencies between each other. If we only generate packets as input test cases, much code about handling syscalls cannot be covered; if we ignore dependencies between system calls and packets, many generated test cases will be meaningless and neglected by TCP stacks without deep processing, which seriously damages fuzzing efficiency. Thus, we need to design a new strategy to generate effective test cases for TCP stacks.

2) Neglect the coverage of state transitions. Existing protocol fuzzing approaches use code coverage as program feedback to cover different protocol states. However, as described in *F2* in Section 2.1, besides states, TCP stacks also have many state transitions that heavily affect TCP execution. Moreover, two test cases covering the same states may cover different state transitions. For example in Figure 5, the test case T1 covers the states S1, S2 and S3 in order, and then the test case T2 covers the states S1, S3 and S2 in order. T1 and T2 both cover the states S1, S2 and S3, and thus existing fuzzing approaches identifies T2 to be useless, as it fails to cover new states. But T1 and T2 cover different state transitions, namely T1 covers S1->S2 and S2->S3 while T2 covers S1->S3 and S3->S2. Thus, T2 is useful in covering new state transitions.

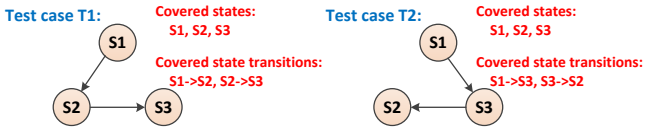

Figure 5: Example of covering states and state transitions.

3) Lack effective detection of semantic bugs. Most existing fuzzing approaches use common bug sanitizers (such as ASan [2] and MSan [39]) to detect memory bugs, such as null-pointer dereferences and use-after-free issues. However, as described in Section 2.2, most of the reported bugs in TCP stacks are semantic bugs, which are not caused by problematic memory accesses. Thus, these bug sanitizers cannot detect semantic bugs in TCP stacks.

3 Key Techniques

To solve the limitations of existing fuzzing in testing TCP stacks, we propose three key techniques: a *dependency-based strategy* to generate effective test cases, a *transition-guided fuzzing approach* to improve the coverage of state transitions and a *differential checker* to detect semantic bugs. We introduce these techniques as follows.

3.1 Dependency-Based Strategy

Inspired by existing two-dimensional fuzzing approaches [29, 64] for file systems, we generate *input sequences* containing syscalls and packets as test cases for TCP stacks. Considering that packets and syscalls accepted by TCP stacks have many dependencies with each other, we design a dependency-based strategy to generate more effective test cases for TCP stacks. Given an original input sequence that improves testing coverage, our strategy mutates it to generate new input sequences. As shown in Figure 6, for each item in the original input sequence, our strategy first selects a mutation type and then mutates this item by considering dependencies with the previously handled items.

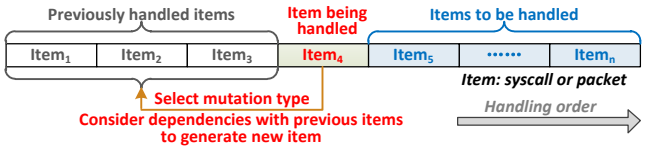

Figure 6: Input sequence mutation.

Mutation-type selection. According to possible operations on a syscall or packet, our strategy provides five available types of mutation (including deletion, addition, replacement and two kinds of changes), as listed in Table 2. Our strategy randomly selects a mutation type to handle each item in the input sequence in order. As a result, different items in the input sequence can be handled with different mutation types.

Item type	Mutation type
Syscall	*Deletion:* delete this syscall.
	Addition: add a new syscall or packet.
	Replacement: replace this syscall with a new packet.
	Change1: change the parameter of this syscall.
	Change2: change the syscall type with the same parameter.
Packet	*Deletion:* delete this packet.
	Addition: add a new syscall or packet.
	Replacement: replace this packet with a new syscall.
	Change1: change the TCP header fields of this packet.
	Change2: change the TCP data length of this packet.

Table 2: Available mutation type.

Dependency-based generation. In Table 2, all of the mutation types except deletion generate a new syscall or packet in the input sequence. As described in F1 in Section 2.1, there are three kinds of dependencies between packets and syscalls. If an input sequence violates these dependencies, it is considered to be invalid and can be simply neglected by TCP stacks. Thus, to generate more effective test cases, our strategy considers these dependencies to generate each item in the input sequence. Specifically, when handling an item, our strategy considers the dependencies between this item and the previously handled items. At present, we have implemented 15 dependency rules in Table 3, by referring to RFC documents (packet-packet and syscall-packet dependencies) and syscall-usage conventions (syscall-syscall dependencies).

Kind	Dependency rule
Syscall-syscall	*SS1:* `socket`, `bind`, `listen` and `accept` are called in order when a connection is passive open.
	SS2: `socket` and `connect` are called in order when a connection is active open.
	SS3: The file descriptor that `socket` or `accept` returns is used by `fcntl`, `ioctl`, `read`, `write` and other syscalls.
	SS4: `read` and `write` can be called only after `accept` or `connect` is called and returns a success.
	SS5: `read` and `write` are never called after `close` is called.
Packet-packet	*PP1:* After a connection is established, the source port and destination port of each packet are fixed.
	PP2: The order and control flags of three-way handshake packets and four-way handshake are never changed.
	PP3: The sequence number of a packet is equal to the sum of sequence number and data length of the previous packet.
	PP4: The timestamps of packets are non-decreasing.
	PP5: The echo reply value in timestamp of a packet is equal to the echo value in timestamp of the previous received packet.
Syscall-packet	*SP1:* `accept` can be called only after the three-way handshake when a connection is passive open.
	SP2: `connect` can be called only before the three-way handshake when a connection is active open.
	SP3: Packets can be sent only after `accept` or `connect` is called and returns a success.
	SP4: The relative acknowledge number of a packet sent to the stack is no more than total length of data sent by `write`.
	SP5: After `close` is called, a packet with the FIN flag should be sent.

Table 3: Implemented dependency rules.

Considering that each TCP stack is implemented according to RFC documents and syscall-usage conventions, we believe that these dependency rules are general to all TCP stacks.

Note that to test whether TCP stacks correctly obey these dependency rules, our strategy also generates some *exceptional input sequences* by deliberately violating packet-packet and syscall-packet dependency rules, with a small probability. Indeed, such input sequences are useful in detecting RFC violations about exception handling.

3.2 Transition-Guided Fuzzing Approach

As described in Section 2.3, code coverage cannot describe state transitions, and thus our fuzzing approach requires a new coverage metric that can effectively describe both states and state transitions.

For a given input sequence, the TCP stack's state is always changed when handling each item (a syscall or packet) in this sequence. Namely, each such item affects the execution situation of the TCP stack. Thus, after handling each item, the TCP stack can be considered to reach a new state. This state can be described with branch coverage (namely the coverage of code branches), as existing fuzzing approaches do. Accordingly, a state transition can be described as the transition between two covered states due to two adjacent input items, namely the change of branch coverage between these input items. Inspired by this idea, we propose a new coverage metric named *branch transition* to describe both states and state transitions. For a given input sequence, a branch transition is represented as a vector that stores both branch coverage for the current input item and the change of branch coverage between the current and previous input items.

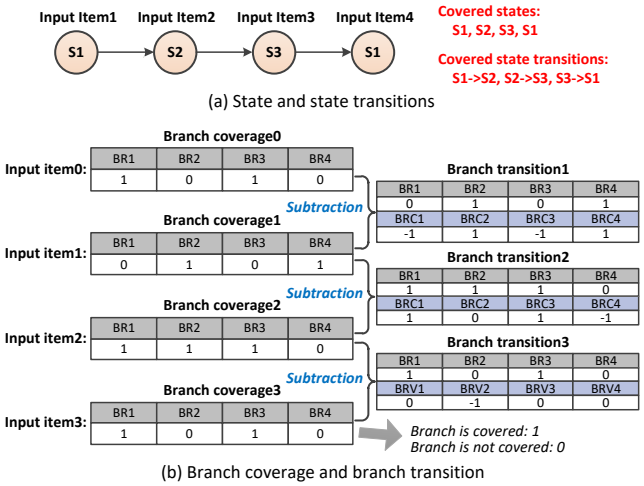

Figure 7: Example of branch transition.

Figure 7 illustrates branch transition using an example. Each state is described using a branch coverage vector, which contains the executed situation (covered or not covered) of each branch in TCP stack code. Then, the state change between the current and previous input items is represented as the subtraction of their branch coverage vectors (*current* − *previous*). Finally, the branch transition of the current input item is obtained as a two-dimensional vector containing its branch coverage vector and the calculated subtraction vector. In Figure 7(a), an input sequence contains four input items which cover the states S1, S2, S3 and S1 in order, and thus it covers three different state transitions S1->S2, S2->S3 and S3->S1. These state transitions are described as three different branch transitions in Figure 7(b). If code coverage is used, *input item3* is identified to be useless, as it covers an old state S1 that is already covered by *input item0*. However, *input item3* actually covers a new state transition S3->S1, which can be successfully described using branch transition.

Our fuzzing approach uses branch transition as program feedback, to effectively cover both states and state transitions. For a given input sequence, if it covers new branch transitions, our fuzzing approach identifies it to be interesting and puts it into the seed corpus for future mutation. Then, our fuzzing approach selects a seed input sequence from the seed corpus and mutates it to generate new input sequences using our dependency-based strategy. We implement most of the fuzzing process by referring to AFL [1].

In fact, besides branch transition, state transition can be also represented as higher-level state change learned by several recent approaches of fuzzing DTLS/TLS protocol implementations [17, 52]. However, the state models learned by these approaches can have mistakes, and thus they still require much manual guidance and validation to ensure correctness. By contrast, branch transition can be automatically and conveniently obtained by collecting runtime information of TCP stacks. Thus, our approach uses branch transition instead of higher-level state change learned by these approaches.

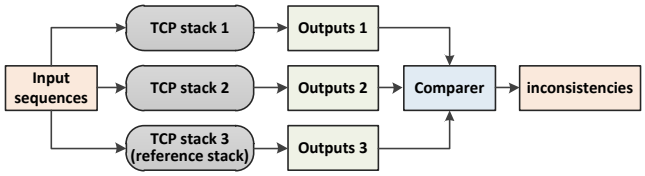

Figure 8: Procedure of our differential checker.

3.3 Differential Checker

To detect semantic bugs, an intuitive solution is to implement semantic checkers by referring to semantic rules in RFC documents. But there are many RFC documents and some semantic rules are even implicit, and thus it is hard to manually implement these checkers. Indeed, different TCP stacks should obey identical semantic rules (many of these rules are defined in RFC documents), and thus they should produce identical or similar outputs for the same inputs. Otherwise, these TCP stacks have implementation inconsistencies, indicating that some of them possibly have semantic bugs.

Based on this idea and inspired by recent approaches of differential testing [11, 12, 65], we design a differential checker for TCP stacks to detect semantic bugs that cause output inconsistencies. As shown in Figure 8, our differential checker provides the same input sequences to multiple TCP stacks, then records their outputs (including return values and parameters of syscalls as well as response packets from TCP stacks), and finally compares these outputs to identify and report inconsistencies. The user can check these inconsistencies to find related semantic bugs.

To improve the efficiency of finding semantic bugs, we suggest using at least one classical and well-tested kernel-level TCP stack (such as Linux TCP or FreeBSD TCP) as a *reference stack* in our differential checker, to test relatively newer TCP stacks. In this case, if our checker reports inconsistencies, it is very likely that one newer TCP stack has semantic bugs.

Our differential checker has three main advantages. First, because different TCP stacks should obey identical semantic rules, the possibility of producing inconsistencies for the same inputs is not large. Thus, the manual work of checking the differences reported by our checker should be much less than that of implementing well-verified checkers of semantic rules. Second, we believe that our checker is also helpful to extracting implicit semantic rules, through identifying and analyzing implementation inconsistencies of multiple TCP stacks. Finally, our checker is scalable and does not introduce runtime overhead for TCP stacks.

At present, our checker records and compares final outputs of TCP stacks, without recording and checking intermediate information (such as window size and packet time) of TCP stacks during packet transmission. Thus, it cannot detect semantic bugs about congestion control and performance. Moreover, our checker detects output inconsistencies between multiple TCP stacks, instead of checking specific RFC doc-

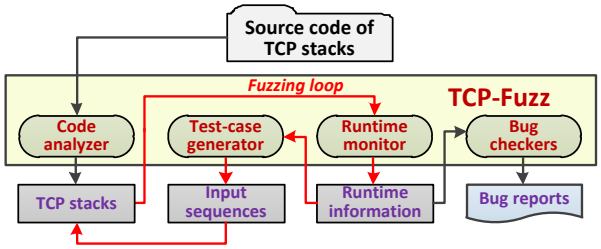

Figure 9: TCP-Fuzz architecture.

uments at runtime. The user needs to manually check RFC documents and analyze the root causes of these inconsistencies, to identify semantic bugs about RFC violations.

4 Framework

Based on the three key techniques in Section 3, we propose a novel fuzzing framework named TCP-Fuzz, to effectively test TCP stacks and detect bugs. We have implemented TCP-Fuzz using Clang 9.0 [13] and Packetdrill [8]. Specifically, we use Clang to perform code instrumentation on TCP stack code, in order to collect covered branches during TCP-stack execution; and we use Packetdrill to send the generated input sequences of syscalls and packets to TCP stacks, and to receive return values and parameters of syscalls as well as response packets from TCP stacks. Overall, TCP-Fuzz consists of four parts:

Code analyzer. It first uses Clang to compile the source code of TCP stacks into LLVM bytecode. Then, it instruments each code branch in the LLVM bytecode. Finally, it compiles the modified LLVM bytecode to generate executable TCP stacks.

Test-case generator. It uses our transition-based fuzzing approach and dependency-based strategy to generate input sequences of syscalls and packets. Each such input sequence is presented as a Packetdrill script, and it is provided to the TCP stacks via Packetdrill. Note that Packetdrill does not support sending some exceptional input sequences that violate dependency rules in Table 3. Thus, we modify Packetdrill by dropping some related checks in its code, to make it support sending such exceptional input sequences.

Runtime monitor. It collects two kinds of runtime information. First, it collects covered branches and calculates branch transitions to provide feedback to our fuzzing approach. Second, it calls Packetdrill interfaces to receive the outputs of each TCP stack, and provides them to our differential checker.

Bug checkers. TCP-Fuzz has three kinds of bug checkers to detect both memory bugs and semantic bugs:

- *Third-party sanitizers.* Existing bug sanitizers (such as ASan [2] and MSan [39]) are used to detect memory bugs by monitoring memory accesses at runtime.
- *Data validator.* We implement this checker to detect semantic bugs leading to incorrect data transfer of TCP stacks, because ensuring data-transfer correctness is a basic property of TCP stacks. Specifically, this checker performs two kinds of validation: (1) whether the data received by the TCP stack via calling `read` is identical to the data stored in packets sent to the TCP stack; (2) whether the data sent from the TCP stack via calling `write` is identical to the data stored in packets received by the remote end.
- *Differential checker.* This checker is used to compare the outputs of multiple TCP stacks for the same inputs, in order to detect semantic bugs.

Deployment. As shown in Figure 10, TCP-Fuzz is deployed in a server-client mode. In this way, TCP-Fuzz can not only use third-party bug sanitizers and data validator in each TCP stack to detect memory bugs and data-correctness-related bugs, but also use the differential checker in multiple TCP stacks to detect their semantic bugs. The TCP-Fuzz server and clients can be deployed in the same machine and communicate with each other via virtual network controllers; or they can be deployed in different machines and communicate with each other via physical network controllers.

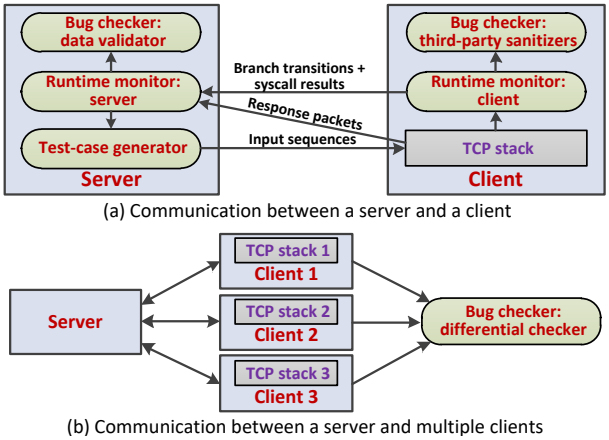

Figure 10: Server-client deployment of TCP-Fuzz.

5 Evaluation

5.1 Experimental Setup

To validate the effectiveness of TCP-Fuzz, we use it to actually test five open-sourced and widely-used TCP stacks, including three user-level ones (TLDK, F-Stack and mTCP) and two kernel-level ones (FreeBSD TCP and Linux TCP). For the three user-level TCP stacks, we test them with the complete fuzzing process of TCP-Fuzz. For the two kernel-level TCP stacks, because they are classical and well-tested, we use them as reference stacks in the differential checker. Moreover, because TCP-Fuzz can only instrument user-level programs at present, we only test the two kernel-level TCP stacks using test cases generated from the user-level TCP stacks, without feedback-driven fuzzing. In the future, we will implement kernel-code instrumentation to support complete fuzzing of kernel-level TCP stacks.

Table 4 shows the basic information about the five tested TCP stacks. Among them, FreeBSD TCP and Linux TCP are two classical kernel-level TCP stacks used in lots of machines; mTCP is a well-known user-level TCP stack in academic community; TLDK and F-Stack are two recent user-level TCP stacks in industry community, and they have been widely deployed in telecom systems and network nodes.

Type	TCP stack	Version	LOC
User-level	TLDK [59]	v2.0	15K
	F-Stack [19]	Commit 8d21adc	25K
	mTCP [26]	Commit 0463aad	18K
Kernel-level	FreeBSD	v12.1	171K
	Linux	v5.6	169K

Table 4: Basic information about tested TCP stacks.

We deploy TCP-Fuzz clients on five regular personal computers, each of which runs a TCP stack to be tested. We deploy TCP-Fuzz server on another personal computer to generate test cases and compare the outputs of these TCP stacks. For each user-level TCP stack, we test it for 48 hours; for each kernel-level TCP stack, we test it by inputting the test cases generated from the three user-level TCP stacks. Besides, we run a third-party sanitizer ASan [2] to detect memory bugs in the user-level TCP stacks.

5.2 Runtime Testing

Table 5 shows the fuzzing results, including covered branches and branch transitions as well as found memory bugs and semantic bugs. Note that TCP-Fuzz does not instrument the two kernel-level TCP stacks, and thus their covered branches and branch transitions are not obtained.

Testing coverage. TCP-Fuzz covers many more branch transitions than branches, indicating that TCP stacks have more state transitions than states during execution. Figure 11 shows the growth of covered branches and branch transitions for the three user-level TCP stacks during fuzzing. Similar to existing fuzzing approaches based on code coverage, TCP-Fuzz covers few new branches during the later tests, but it still covers many new branch transitions during these tests.

Found bugs. TCP-Fuzz finds 56 real bugs in the five tested TCP stacks, including 8 memory bugs and 48 semantic bugs. We reported these bugs to related developers, and 40 of them have been confirmed. We are still waiting for responses for the remaining bugs (for example, the mTCP code in github has not been updated for a long time, and thus we have not received any response to our reported bugs in mTCP). Besides, 23 of the confirmed bugs have been fixed.

Output inconsistencies. TCP-Fuzz reports 15.1K inconsistencies between the five tested TCP stacks, and we analyze their root causes to identify semantic bugs, through our manual review of RFC documents and observation of TCP stack execution. Similar to SQLancer [51] and libFuzzer [32], for inconsistencies that we identify as semantic bugs, we manually

Stack	Testing coverage		Found bugs	
	Branch	*Transition*	*Memory / Semantic*	*Confirmed / Fixed*
TLDK	1.3K	329.4K	2 / 26	28 / 19
F-Stack	7.5K	46.8K	1 / 6	6 / 1
mTCP	1.2K	47.9K	5 / 9	0 / 0
FreeBSD	-	-	0 / 6	5 / 2
Linux	-	-	0 / 1	1 / 1
Total	10.0K	424.1K	8 / 48	40 / 23

Table 5: Results of fuzzing TCP stacks.

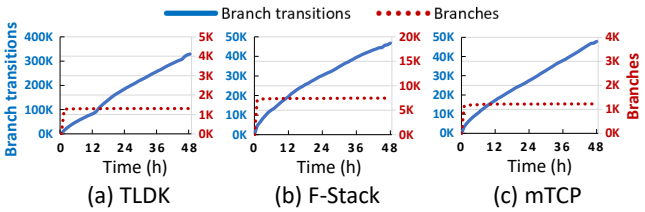

Figure 11: Covered branches and branch transitions.

fix them using the developers' patches or by ourselves, to reduce related inconsistencies. We iteratively repeat this process until inconsistencies never occur, to count unique semantic bugs, resulting in 48 semantic bugs. We observe that many output inconsistencies are repeated, as they are triggered by the same root cause. Only one output inconsistency is considered to be benign. Specifically, when normal packets come after a FIN packet and their sequence numbers are larger than that of the FIN packet, FreeBSD, F-Stack and mTCP drop the FIN packet, while Linux TCP and TLDK reset the connection. As RFC documents do not stipulate how to handle this case, we are not sure which strategies are correct.

Bug-finding process. We also analyze how TCP-Fuzz finds these 56 bugs, and show the results in Table 6. The 8 memory bugs are all found by ASan, 2 semantic bugs are found by the data validator and 46 semantic bugs are found by the differential checker. The results indicate that our differential checker is effective in finding semantic bugs. Besides, we also highlight that 2 semantic bugs found by the data validator are quite dangerous, because they directly cause TCP stacks to send or receive incorrect data, badly damaging data-transfer correctness. Moreover, 28 bugs are found via exceptional input sequences generated by deliberately violating the dependency rules listed in Table 3, while 28 bugs are found via normal input sequences generated by obeying these rules. The results indicate that the TCP stack code about handling exceptional inputs is error-prone in practice. Thus, exception handling in TCP stacks should receive more attention in testing.

Stack	Number of tests	Checker			Input sequence type	
		ASan	*Data*	*Differential*	*Exceptional*	*Normal*
TLDK	123K	2	1	25	11	17
F-Stack	128K	1	1	5	6	1
mTCP	170K	5	0	9	4	10
FreeBSD	421K	0	0	6	6	0
Linux	421K	0	0	1	1	0
Total	1,263K	8	2	46	28	28

Table 6: Statistics of bug-finding process.

Stack	RFC violation	Syscall issues	Implicit rules
TLDK	15	9	2
F-Stack	5	1	0
mTCP	7	2	0
FreeBSD	5	1	0
Linux	1	0	0
Total	33	13	2

Table 7: Root causes of semantic bugs.

RFC document	*791*	*793*	*1122*	*5961*	*6093*	*6691*	*7323*	*7413*
Semantic bug	2	7	1	6	1	1	12	3

Table 8: Distribution of RFC violations.

Root causes of memory bugs. For the 8 found memory bugs, 2 are use-after-free issues, 3 are null-pointer dereferences, 2 are buffer-overflow issues, and 1 is a division-by-zero issue.

Root causes of semantic bugs. For the 48 found semantic bugs, we summarize three root causes in Table 7 and find that:

(1) 33 semantic bugs are RFC violations, and they violate semantic rules explicitly described in the 8 RFC documents shown in Table 8. For example, 6 semantic bugs (3 in TLDK, 1 in mTCP, 1 in F-Stack and 1 in FreeBSD TCP) are RFC 5961 [48] violations. Indeed, the RFC 5961 document is designed to mitigate the influence of blind in-window attacks [36] by changing the range of acceptable sequence numbers in reset packets and acknowledge numbers in normal packets. Thus, these semantic bugs can be exploited by attackers to reset the connection [62] or inject malicious data [7, 10, 45] via blind in-window attacks.

(2) 13 semantic bugs are caused by incorrect results of syscalls. For example, 3 semantic bugs (1 in TLDK, 1 in F-Stack and 1 in FreeBSD TCP) are caused by using an invalid file descriptor obtained from `socket` after `listen` and `accept` are called in order. Indeed, after `listen` and `accept` are called, the previous file descriptor obtained from `socket` becomes invalid, and thus `ioctl`, `read` and `write` should return an error code when using this file descriptor. However, TLDK, F-Stack and FreeBSD TCP return zero to indicate a success in this case, causing semantic bugs.

(3) 2 semantic bugs in TLDK are caused by violating implicit semantic rules. Specifically, RFC documents do not describe how to handle unknown options in the TCP packet header. In our tests, F-Stack, mTCP, FreeBSD TCP and Linux TCP simply ignore these options and accept related packets, but TLDK drops related packets or enters an infinite loop when handling these options.

5.3 Influences of the Found Bugs

We manually review the 56 found bugs to estimate their influences on the reliability and security of TCP stacks. The results are shown in Table 9. We find that 6 semantic bugs about RFC 5961 violations are vulnerable to the MIMT (Man-in-the-middle) attacks; 9 bugs (including 4 memory bugs and 5 semantic bugs) can cause data corruption; 8 bugs (including 4 memory bugs and 4 semantic bugs) can cause crashes or denial of services; 21 semantic bugs can cause functional errors of data communication; and 12 semantic bugs can reduce the efficiency of data communication.

Stack	*MIMT attack*	*Corruption*	*Crash/DoS*	*Functional error*	*Inefficiency*
TLDK	3	6	4	12	3
F-Stack	1	1	0	3	2
mTCP	1	2	4	3	4
FreeBSD	1	0	0	3	2
Linux	0	0	0	0	1
Total	6	9	8	21	12

Table 9: Reliability and security influence of the found bugs.

Figure 12 shows three bugs found by TCP-Fuzz, including 1 memory bug and 2 semantic bugs. This figure also shows the related test cases in form of Packetdrill scripts generated by TCP-Fuzz for finding these bugs.

Use-after-free issue in TLDK. In Figure 12(a), the function `rx_fin` free the data of `s->tx.q` by calling `empty_tq`. Then, this data is used by accessing `m->data_len` in the function `txq_rst_nxt_head`, causing a use-after-free issue. Once this bug is triggered, attackers can modify the data of `s->tx.q` to inject malicious data in the TCP connection. To fix this bug, the developer submits a patch to assign zero to `s->tcb.snd.una_offset` in the function `rx_fin`, in order to avoid accessing the data of `s->tx.q` in the function `txq_rst_nxt_head`.

RFC 7323 violation in FreeBSD TCP. In Figure 12(b), as shown in the annotation of the function `syncache_expand`, if timestamps are not negotiated in the first two packets of three-way handshake, FreeBSD TCP rejects the third packet containing the timestamp option and resets the TCP connection. However, the RFC 7323 document stipulates that the third packet in this case should be normally accepted. Once this bug is triggered, the TCP connection can be abnormally disconnected during three-way handshake, causing a functional error. To fix this bug, the developer submits a patch to modify the problematic code according to the related semantic rule in the RFC 7323 document.

RFC 793 violation in mTCP. In Figure 12(c), if the sequence number of the current packet is smaller than the next expected sequence number, mTCP drops the current packet. However, the RFC 793 document stipulates if the current packet overlaps the range of the expected receive window, this packet should be accepted. Once this bug is triggered, data communication can be inefficient due to abnormally dropping packets. To fix this bug, our preliminary solution is to accept the content of the current packet within the range of expected receive window.

5.4 Comparison to Existing Fuzzing Tools

We perform the comparison in two ways. First, we compare TCP-Fuzz to two classical and widely-used fuzzing approaches, namely AFL [1] and Syzkaller [57]. Considering

Generated test case in form of a Packetdrill script

```
.....
+0 write(4, ..., 5000) = 5000
+0.1 > . 1:5001(5000) ack 1
+0 < . 1:1(0) ack 1001 win 257
+0.1 close(4) = 0
+0.1 < F. 2001:2001(0) ack 5002 win 257
+0.5 < F. 1:2001(2000) ack 5002 win 257 // Use after free!
```

FILE: TLDK/src/lib/libtle_l4p/tcp_rxtx.c

```
int rx_fin(...) {
    .....
    if (s->tcb.snd.fss >= s->tcb.snd.nxt &&
        si->ack == (uint32_t)s->tcb.snd.fss) {
        .....
+       s->tcb.snd.una_offset = 0;
        empty_tq(s);  // FREE the data of s->tx.q
    }
    .....
}
```

FILE: TLDK/src/lib/libtle_l4p/tcp_txq.h

```
void tcp_txq_rst_nxt_head(...) {
    struct rte_ring *r = s->tx.q;
    .....
    offset = s->tcb.snd.una_offset;
    if (offset) {
        m = (struct rte_mbuf*)
            (_rte_ring_get_data(r)[r->cons.tail & r->mask]);
        data_len = m->data_len - PKT_L234_HLEN(m);  // USE
        .....
    }
    .....
}
```

(a) Use-after-free issue in TLDK

Generated test case in form of a Packetdrill script

```
.....
+0 < S 0:0(0) win 12336
+0 ~ +1  > S. 0:0(0) ack 1 <mss 1460>
+0.1 < . 1:1(0) ack 1 win 25710 <TS val 100 ecr 0,eol,eol>
+0.1 > R 1:1(0) win 0 // RST is sent?
```

FILE: FreeBSD/src/sys/netinet/tcp_syncache.c

```
int syncache_expand(...) {
    .....
+   // If timestamps were not negotiated during SYN/ACK and a
+   // segment with a timestamp is received, ignore the
+   // timestamp and process the packet normally.
+   // See section 3.2 of RFC 7323.
+   if (!(sc->sc_flags & SCF_TIMESTAMP) &&
+       (to->to_flags & TOF_TS)) {
+       if ((s = tcp_log_addrs(inc, th, NULL, NULL))) {
+           log(LOG_DEBUG, ...);
+           free(s, M_TCPLOG);
+           s = NULL;
+       }
+   }
    .....
-   // If timestamps were not negotiated during SYN/ACK they
-   // must not appear on any segment during this session.
-   if (!(sc->sc_flags & SCF_TIMESTAMP) &&
-       (to->to_flags & TOF_TS)) {
-       if ((s = tcp_log_addrs(inc, th, NULL, NULL)))
-           log(LOG_DEBUG, ...);
-       goto failed;
-   }
    .....
}
```

(b) RFC 7323 violation in FreeBSD TCP

Generated test case in form of a Packetdrill script

```
.....
+0 < . 1:1001(1000) ack 1
+0.1 read(4, ..., 1000) =  1000
+0 < . 501:1501(1000) ack 1
+0.1 read(4, ..., 500) =  500 // returns 0?
```

FILE: mtcp/src/tcp_ring_buffer.c

```
int RBPut(...) {
    .....
+   if (cur_seq < buff->head_seq) {
+       len -= buff->head_seq - cur_seq;
+       data += buff->head_seq - cur_seq;
+       cur_seq = buff->head_seq;
+   }
    .....
    if (GetMinSeq(buff->head_seq, cur_seq) != buff->head_seq)
        return 0;
    .....
}
```

(c) RFC 793 violation in mTCP

Figure 12: Example bugs found by TCP-Fuzz.

that AFL and Syzkaller cannot directly test TCP stacks, we implement a AFL-like and a Syzkaller-like fuzzing tool for comparison. Specifically, the AFL-like tool only generates packet sequences according to code coverage, by considering dependencies between packets; the Syzkaller-like tool only generates syscall sequences according to code coverage, by considering dependencies between syscalls. The two fuzzing tools use the three bugs checkers used by TCP-Fuzz. Second, we compare TCP-Fuzz to three state-of-the-art and open-sourced protocol fuzzing approaches, namely Boofuzz [6], Fuzzotron [20] and AFLNet [43]. The three fuzzing tools use ASan to detect memory bugs. In the experiments, we select TLDK as the target. Indeed, TLDK contains a half of all bugs found by TCP-Fuzz, and thus the compared approaches are more likely to find bugs in TLDK than the other TCP stacks.

AFL-like and Syzkaller-like tools. Figure 13 plots the covered branch transitions of the two fuzzing tools and TCP-Fuzz. We find that TCP-Fuzz covers many more branch transitions than the two fuzzing tools. It indicates that generating two-dimensional test cases (syscalls and packets) is more effective in improving testing coverage than generating only one-dimensional test cases (syscalls or packets). Besides, we also find that the AFL-like tool covers more branch transitions than Syzkaller-like tool, indicating that the state transitions of TCP stacks are more sensitive to packets than to syscalls. Moreover, as shown in Figure 14, the AFL-like and Syzkaller-like tools find 7 and 4 bugs, respectively. TCP-Fuzz finds all these bugs, and it also finds 17 bugs missed by the two fuzzing tools, due to covering more branch transitions.

Existing fuzzing tools of network protocols. As shown in Figure 14, Boofuzz finds 1 null-pointer dereference, and Fuzzotron and AFLNet do not find any bug. Indeed, the three fuzzing tools only generate packets according to code coverage, without considering the dependencies between packets.

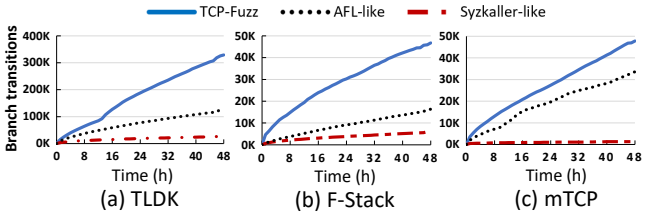

Figure 13: Comparison of testing coverage.

Thus, they miss many useful state transitions due to ineffective test-case generation and limited program feedback. Fuzzotron and AFLNet are mainly used to test implementations of application-layer network protocols, so their ability to test transport-layer TCP stacks is weak. TCP-Fuzz finds the null-pointer dereference found by Boofuzz, and it also finds 27 bugs (including 1 memory bug) missed by the three fuzzing tools, due to using the three key techniques in Section 3.

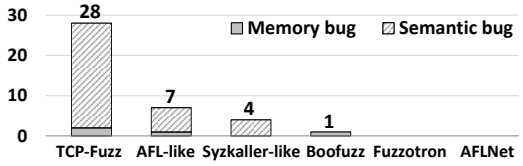

Figure 14: Comparison of found bugs in TLDK.

6 Discussion

Differential checker. In our evaluation, this checker reports 15.1K inconsistencies between the five tested TCP stacks. We intended to design an automated tool to count unique bugs from these inconsistencies, but we found that doing so is difficult for two reasons. First, these inconsistencies are obtained from input sequences of syscalls and packets, and it is hard to automatically identify whether two such sequences have identical semantic information. Second, understanding

the root causes of these inconsistencies requires TCP-specific knowledge that is hard to extract as fixed patterns.

Testing congestion control. Congestion control is an important functionality for TCP stacks, but TCP-Fuzz cannot test it at present, due to ignoring congestion-control-related packet information (such as the length of accepted data for each packet) and code information (such as the variables about congestion window size). In the future, we will collect and check such information by referring to related work [27, 56], to test congestion control implementations of TCP stacks.

Limitations and future works. TCP-Fuzz can be strengthened in some aspects. First, as described in Section 5.1, TCP-Fuzz cannot instrument kernel code in the current implementation, and thus it fails to completely test kernel-level TCP stacks. To solve this limitation, we plan to perform kernel-code instrumentation or tune existing VM-based approaches [25, 55] in TCP-Fuzz. Second, TCP-Fuzz fails to record intermediate information (such as window size and packet time) of TCP stacks during packet transmission, and thus it cannot detect semantic bugs about congestion control and performance. To solve this limitation, we plan to record such intermediate information and implement related checkers to detect these semantic bugs. Third, TCP-Fuzz fails to check concurrent memory accesses, and thus it cannot find concurrency bugs in TCP stacks. To solve this limitation, we plan to introduce existing concurrency-analysis approaches [18, 31] to detect concurrency bugs in TCP stacks. Finally, QUIC [46] is a new and promising transport-layer network protocol proposed, and it is expected to replace TCP in the future. Thus, we also plan to extend TCP-Fuzz to testing QUIC implementations.

7 Related Work

7.1 Network Protocol Fuzzing

Fuzzing is a popular testing technique to detect bugs in software systems. Many fuzzing approaches have been proposed to test the implementations of network protocols.

Some approaches [6, 37, 38, 54, 60] use grammar-based fuzzing. They utilize hard-coded or user-defined grammar specifications to guide test-case generation. These specifications define data structure or field types of packets to be generated. For example, TLS-Attacker [54] is a flexible TLS testing framework for developers to test their TLS implementations by writing Java code or XML-based specifications.

Several recent approaches [17, 43, 52] perform stateful protocol fuzzing. AFLNet [43] can learn basic state models of network protocols to improve seed selection and mutation. Fiterau-Brostean et al. [17] propose a practical tool by extending TLS-Attacker [54], to learn comprehensive state models of multiple DTLS implementations. By comparing these learned state models, the user can infer vulnerabilities in DTLS implementations.

However, these approaches are limited in testing TCP stacks. First, these approaches only generate packets as test cases, but TCP stacks receive both packets and syscalls as inputs, and thus these approaches may miss much code for handling syscalls. Second, these approaches use code coverage as program feedback to cover states, but code coverage cannot effectively describe state transitions in TCP state model. Finally, many of these approaches only use existing bug sanitizers to detect memory bugs, but fail to detect semantic bugs. To solve these limitations, TCP-Fuzz uses a dependency-based strategy to generate effective test cases of packets and syscalls, a transition-guided fuzzing approach to improve the coverage of state transitions, and a differential checker to detect semantic bugs of TCP stacks.

7.2 TCP Stack Checking

Packetdrill [8] is a scripting tool to test the correctness and performance of network stacks. The user can write tcpdump-like scripts to generate and maintain test cases for new feature development and regression testing of network stacks. But writing effective test cases in form of Packetdrill scripts requires a deep understanding of TCP stacks and much manual effort. To solve this problem, TCP-Fuzz automatically generates Packetdrill scripts as test cases according to program feedback and dependencies between packets and syscalls.

Some approaches [24, 34, 40, 41, 53] perform model checking or formal verification of TCP stacks. For example, Lockefeer et al. [34] use μCRL and LTSmin [35] toolsets to generate state spaces and perform formal verification of TCP extended with the Window Scale Option. Hoque et al. [24] use symbolic execution to precisely simulate program execution with symbolic inputs and explore all possible execution paths, and also use an off-the-shelf model checker to check temporal properties of TCP stacks. But these approaches require much manual effort and TCP-specific knowledge to provide a complete and correct TCP state model, and they are often time-consuming due to high complexity of TCP state transitions.

Some approaches [9, 30] perform static analysis of TCP stack source code. For example, PacketGuardian [9] uses static taint analysis to check the packet handling logic of various network protocol implementations, to detect packet-injection vulnerabilities. However, these approaches often introduce false positives in practice, due to lacking exact runtime information for analysis.

Some approaches [3–5, 66] analyze execution traces of TCP stacks to infer RFC violations. For example, Bishop et al. [4] analyze the execution traces with higher-order logic specifications, to identify differences between multiple network protocols stacks and thus to detect possible RFC violations. However, these approaches require substantial and effective test cases to achieve high testing coverage. To solve this problem, TCP-Fuzz automatically generates effective test cases with fuzzing.

Some approaches [27, 28] perform runtime testing for automated attack discovery of TCP stacks. These approaches strategically generate packets to cover different TCP states, which are tracked according to packet information and predefined protocol state machines, without modifying TCP stack code. Different from these approaches, TCP-Fuzz generates both packets and syscalls as test cases, with the guidance of branch transition; TCP-Fuzz does not require pre-defined protocol state machines, but it performs code instrumentation on TCP stacks to collect branch transition.

7.3 Differential Testing

To find semantic bugs, many approaches [11, 12, 23, 42, 47, 61, 65] perform differential testing to identify implementation inconsistencies between multiple programs of the same functionalities. Classfuzz [12] and Classming [11] syntactically mutate Java bytecode files and execute them on different JVM implementations, to identify their inconsistencies. The two approaches both use Markov Chain Monte Carlo (MCMC) sampling to guide mutator selection to improve test-case generation. C2V [65] uses randomized differential testing to detect bugs in code coverage tools (such as gcov and llvm-cov). It randomly generates program code files and compares coverage reports of code coverage tools to identify inconsistencies. Inspired by these approaches, we design a useful differential checker to detect semantic bugs in TCP-stack fuzzing.

8 Conclusion

In this paper, we develop a novel fuzzing framework named TCP-Fuzz, to effectively test TCP stacks and detect bugs. It uses three key techniques: (1) a dependency-based strategy that considers dependencies between packets and system calls, to generate effective test cases; (2) a transition-guided fuzzing approach that uses branch transition as program feedback, to improve the coverage of state transitions; (3) a differential checker that compares the outputs of multiple TCP stacks for the same inputs, to detect semantic bugs. We have evaluated TCP-Fuzz on five widely-used TCP stacks, and find 56 real bugs (including 8 memory bugs and 48 semantic bugs). We also compare TCP-Fuzz to existing fuzzing approaches, and it finds many real bugs missed by these approaches.

In the future, we plan to improve TCP-Fuzz to detect congestion control issues and performance problems, and to apply TCP-Fuzz to other TCP stacks and QUIC implementations.

Acknowledgment

We thank our shepherd, Cristina Nita-Rotaru, and anonymous reviewers for their helpful advice on the paper. We also thank the developers of TCP stacks, who gave useful feedback and advice to us. This work was supported by the Natural Science Foundation of China under Project 62002195 and the China Postdoctoral Science Foundation under Project 2019T120093. Jia-Ju Bai is the corresponding author.

References

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[2] ASan: address sanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[3] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: rigorous test oracle specification and validation for TCP/IP and the Sockets API. *Journal of the ACM*, 66(1):1:1–1:77, 2018.

[4] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proceedings of the ACM SIGCOMM 2005 Conference*, pages 265–276, 2005.

[5] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proceedings of the 33rd International Symposium on Principles of Programming Languages (POPL)*, pages 55–66, 2006.

[6] Boofuzz: network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz.

[7] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path TCP exploits: global rate limit considered dangerous. In *Proceedings of the 25th USENIX Security Symposium*, pages 209–225, 2016.

[8] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-Keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: scriptable network stack testing, from sockets to packets. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 213–218, 2013.

[9] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. Static detection of packet injection vulnerabilities: a case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd International Conference on Computer and Communications Security (CCS)*, pages 388–400, 2015.

[10] Weiteng Chen and Zhiyun Qian. Off-path TCP exploit: how wireless routers can jeopardize your secrets. In *Proceedings of the 27th USENIX Security Symposium*, pages 1581–1598, 2018.

[11] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 1257–1268, 2019.

[12] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th International Conference on Programming Language Design and Implementation (PLDI)*, pages 85–99, 2016.

[13] Clang: a LLVM-based compiler for C/C++ program. https://clang.llvm.org/.

[14] FreeBSD commit bc35229fad1f: add PAWS check for ACK segments in syncache code. https://gitlab.com/FreeBSD/freebsd-src/commit/bc35229fad1f.

[15] CVE-2019-11478. https://nvd.nist.gov/vuln/detail/CVE-2019-11478.

[16] Aled Edwards and Steve Muir. Experiences implementing a high performance TCP in user-space. *ACM SIGCOMM Computer Communication Review*, 25:196–205, 1995.

[17] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, pages 2523–2540, 2020.

[18] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, pages 215–228, 2011.

[19] F-Stack: high performance network framework based on DPDK. http://www.f-stack.org.

[20] Fuzzotron: a network fuzzer supporting TCP, UDP an multithreading. https://github.com/denandz/fuzzotron.

[21] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: stateful black-box fuzzing of proprietary network protocols. In *Proceedings of the 11th International Conference on Security and Privacy in Communication Systems, Security and Privacy in Communication Networks*, pages 330–347, 2015.

[22] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security (IJCSNS)*, 10(8):239, 2010.

[23] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 International Symposium on Foundations of Software Engineering (FSE)*, pages 739–743, 2018.

[24] Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN)*, pages 627–638, 2017.

[25] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 754–768, 2019.

[26] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 489–502, 2014.

[27] Samuel Jero, Md. Endadul Hoque, David R. Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[28] Samuel Jero, Hyojeong Lee, and Cristina Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *Proceedings of the 45th International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2015.

[29] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th International Symposium on Operating Systems Principles (SOSP)*, pages 147–161, 2019.

[30] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding protocol manipulation attacks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 26–37, 2011.

[31] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th International Symposium on Operating Systems Principles (SOSP)*, pages 162–180, 2019.

[32] libFuzzer: a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[33] LLVM compiler infrastructure. https://llvm.org/.

[34] Lars Lockefeer, David M. Williams, and Wan J. Fokkink. Formal specification and verification of TCP extended with the window scale option. *Science of Computer Programming (SCP)*, 118:3–23, 2016.

[35] LTSmin: model checking and minimization of labelled transition systems. https://ltsmin.utwente.nl/.

[36] Matthew Luckie, Robert Beverly, Tiange Wu, Mark Allman, and kc claffy. Resilience of deployed TCP to blind attacks. In *Proceedings of the 2015 Internet Measurement Conference (IMC)*, pages 13–26, 2015.

[37] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. Polar: function code aware fuzz testing of ICS protocol. *ACM Transactions on Embedded Computing Systems*, 18(5s):93:1–93:22, 2019.

[38] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. ICS protocol fuzzing: coverage guided packet crack and generation. In *Proceedings of the 57th International Design Automation Conference (DAC)*, pages 1–6, 2020.

[39] MSan: memory sanitizer. https://github.com/google/sanitizers/wiki/MemorySanitizer.

[40] Sandra L. Murphy and A. Udaya Shankar. Service specification and protocol construction for the transport layer. In *Proceedings of the ACM SIGCOMM 1988 Conference*, pages 88–97, 1988.

[41] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 155–168, 2004.

[42] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. HyDiff: hybrid differential software analysis. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 1273–1285, 2020.

[43] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: a greybox fuzzer for network protocols. In *Proceedings of the 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465, 2020.

[44] Lei Qian and Brian E. Carpenter. A flow-based performance analysis of TCP and TCP applications. In *Proceedings of the 18th International Conference on Networks (ICON)*, pages 41–45, 2012.

[45] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 19th International Conference on Computer and Communications Security (CCS)*, pages 593–604, 2012.

[46] QUIC: a multiplexed stream transport over UDP. https://www.chromium.org/quic.

[47] Gaganjeet Singh Reen and Christian Rossow. DPIFuzz: a differential fuzzing framework to detect DPI elusion strategies for QUIC. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, pages 332–344, 2020.

[48] RFC 5961: improving TCP's robustness to blind in-window attacks. https://tools.ietf.org/html/rfc5961.

[49] RFC 7323: TCP extensions for high performance. https://tools.ietf.org/html/rfc7323.

[50] RFC 793: TCP (Transmission Control Protocol). https://tools.ietf.org/html/rfc793.

[51] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th International Symposium on Operating Systems Design and Implementation (OSDI)*, pages 667–682, 2020.

[52] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *Proceedings of the 24th USENIX Security Symposium*, pages 193–206, 2015.

[53] Mark Anthony Shawn Smith. *Formal verification of TCP and T/TCP*. PhD thesis, Massachusetts Institute of Technology, 1997.

[54] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)*, pages 1492–1504, 2016.

[55] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the 29th USENIX Security Symposium*, pages 2541–2557, 2020.

[56] Wei Sun, Lisong Xu, and Sebastian Elbaum. Scalably testing congestion control algorithms of real-world TCP implementations. In *Proceedings of the 2018 International Conference on Communications (ICC)*, pages 1–7, 2018.

[57] Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[58] Possible TCP options. https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml.

[59] TLDK: Transport Layer Development Kit in network. https://github.com/FDio/tldk/.

[60] Andreas Walz and Axel Sikora. Exploiting dissent: towards fuzzing-based differential black-box testing of TLS implementations. *IEEE Transactions Dependable Secure Computing (TDSC)*, 17(2):278–291, 2020.

[61] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. SymTCP: eluding stateful deep packet inspection with automated discrepancy discovery. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[62] Paul Watson. Slipping in the window: TCP reset attacks. *Technical Whitepaper*, 2004.

[63] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of caching strategies in modern cellular backhaul networks. In *Proceeding of the 11th International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 319–332, 2013.

[64] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 818–834, 2019.

[65] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 488–499, 2019.

[66] Yanyan Zhuang, Eleni Gessiou, Steven Portzer, Fraida Fund, Monzur Muhammad, Ivan Beschastnikh, and Justin Cappos. Netcheck: network diagnoses from black-box traces. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 115–128, 2014.