Linear-time Temporal Logic guided Greybox Fuzzing

Ruijie Meng*
National University of Singapore
Singapore
ruijie@comp.nus.edu.sg

Zhen Dong*†‡
Fudan University
China
zhendong@fudan.edu.cn

Jialin Li
National University of Singapore
Singapore
lijl@comp.nus.edu.sg

Ivan Beschastnikh‡
University of British Columbia
Canada
bestchai@cs.ubc.ca

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Software model checking as well as runtime verification are verification techniques which are widely used for checking temporal properties of software systems. Even though they are property verification techniques, their common usage in practice is in "bug finding", that is, finding violations of temporal properties. Motivated by this observation and leveraging the recent progress in fuzzing, we build a greybox fuzzing framework to find violations of Linear-time Temporal Logic (LTL) properties.

Our framework takes as input a sequential program written in C/C++, and an LTL property. It finds violations, or counterexample traces, of the LTL property in stateful software systems; however, it does not achieve verification. Our work substantially extends directed greybox fuzzing to witness arbitrarily complex event orderings. We note that existing directed greybox fuzzing approaches are limited to witnessing reaching a location or witnessing simple event orderings like use-after-free. At the same time, compared to model checkers, our approach finds the counterexamples faster, thereby finding more counterexamples within a given time budget.

Our LTL-Fuzzer tool, built on top of the AFL fuzzer, is shown to be effective in detecting bugs in well-known protocol implementations, such as OpenSSL and Telnet. We use LTL-Fuzzer to reproduce known vulnerabilities (CVEs), to find 15 zero-day bugs by checking properties extracted from RFCs (for which 12 CVEs have been assigned), and to find violations of both safety as well as liveness properties in real-world protocol implementations. Our work represents a practical advance over software model checkers — while simultaneously representing a conceptual advance over existing greybox fuzzers. Our work thus provides a starting point for understanding the unexplored synergies among software model checking, runtime verification and greybox fuzzing.

*Joint first authors
†Corresponding Author
‡Zhen Dong and Ivan Beschastnikh participated in the work while being at the National University of Singapore as post-doc and visiting Associate Professor respectively.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510082

ACM Reference Format:
Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-time Temporal Logic guided Greybox Fuzzing. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510082

1 INTRODUCTION

Software model checking is a popular validation and verification method for reactive stateful software systems. It is an automated technique to check temporal logic properties (constraining event orderings in program execution) against a finite state transition system. Model checking usually suffers from the state space explosion problem; this is exacerbated in software systems which are naturally infinite-state. To cope with infinitely many states, the research community has looked into automatically deriving a hierarchy of finite state abstractions via predicate abstractions and abstraction refinement of the program's data memory (e.g. see [13]). Whenever a counterexample trace is found in such model checking runs, the trace can be analyzed to find (a) whether it is a spurious counterexample introduced due to abstractions, or (b) the root-cause / bug causing the counterexample. This has rendered model checking to be a useful automated bug finding method for software systems.

Runtime verification is a lightweight and yet rigorous verification method, which complements model checking [15, 47, 48]. In runtime verification, a single execution of a system is dynamically checked against formally specified properties (e.g, temporal logic properties). Specifically, formal properties specify the correct behaviours of a system. Then the system is instrumented to capture events that are related to the properties being checked. During runtime, a monitor collects the events to generate execution traces and checks whether the traces conform to the specified properties. When the properties are violated, it reports violations. Runtime verification aims to achieve a lightweight but not full-fledged verification method. It verifies software systems at runtime without the need of constructing models about software systems and execution environments. However, to generate effective execution traces, software systems are required to be fed many inputs. These inputs are usually obtained manually or via random generation [47]; therefore, runtime verification may take much manual effort and explore many useless inputs in the process of exposing property violations.

Parallel to the works in software model checking and runtime verification, greybox fuzzing methods [1, 3] have seen substantial recent advances. These methods conduct a biased random search over the domain of program inputs, to find bugs or vulnerabilities. The main advantage of greybox fuzzing lies in its scalability to large software systems. However, greybox fuzzing is only a testing (not verification) method and it is mostly useful for finding witnesses to simple oracles such as crashes or overflows. Recently there have been some extension of greybox fuzzing methods towards generating witnesses of more complex oracles, such as tests reaching a location [17]. However, generating inputs and traces satisfying a complex temporal property remains beyond the reach of current greybox fuzzing tools. Thus, today's greybox fuzzing technology cannot replace the bug-finding abilities of software model checking and runtime verification.

In this paper, we take a step forward in understanding the synergies among software model checking, runtime verification and greybox fuzzing. Given a sequential program and a Linear-time Temporal Logic (LTL) property ϕ, we construct the Büchi automata $\mathcal{A}_{\neg\phi}$ accepting $\neg\phi$, and use this automata to guide the fuzz campaign. Thus, given a random input exercising an execution trace π, we can check the "progress" of π in reaching the accepting states of $\mathcal{A}_{\neg\phi}$, and derive from $\mathcal{A}_{\neg\phi}$, the events that are needed to make further progress in the automata. Furthermore, in general, traces accepted by $\mathcal{A}_{\neg\phi}$ are infinite in length and visit an accepting state infinitely often. To accomplish the generation of such infinite-length traces in the course of a fuzz campaign, we can take application state snapshots (at selected program locations) and detect whether an accepting state of $\mathcal{A}_{\neg\phi}$ is being visited with the same program state. The application state snapshot can also involve a state abstraction if needed, in which case the counterexample trace can be subsequently validated via concrete execution.

We present a fuzzing-based technique that directs fuzzing to find violations of *arbitrary LTL properties*. To the best of our knowledge, no existing fuzzing technique is capable of finding violations of complex constraints on event orderings such as LTL properties. Existing works on greybox fuzzing are limited to finding witnesses of simple properties such as crashes or use-after-free. This is the main contribution of our work: algorithms and an implementation of our ideas in a tool that is able to validate *any LTL property*, thereby covering a much more expressive class of properties than crashes or use-after-free. Our work adapts directed greybox fuzzing (which directs the search towards specific program locations) to find violations of temporal logic formulae. We realize our approach for detecting violations of LTL properties in a new greybox fuzzer tool called LTL-Fuzzer. LTL-Fuzzer is built on top of the AFL fuzzer [1] and involves additional program instrumentation to check if a particular execution trace is accepted by the Büchi automaton representing the negation of the given LTL property.

We evaluated LTL-Fuzzer on well-known and large-scale protocol implementations such as OpenSSL, OpenSSH, and Telnet. We show that it efficiently finds bugs that are violations of both safety and liveness properties. We use LTL-Fuzzer to reproduce known bugs/violations in the protocol implementations. More importantly, for 50 LTL properties that we manually extracted from Request-for-Comments (RFCs), LTL-Fuzzer found 15 new bugs (representing the violation of these properties), out of which 12 CVEs have been assigned. These are zero-day bugs which have previously not been found. We make the data-set of properties and the bugs found available with this paper. We expect that in future, other researchers will take forward the direction in this paper to detect temporal property violations via greybox fuzzing. The dataset of bugs found by LTL-Fuzzer can thus form a baseline standard for future research efforts. The dataset and tool are available at **https://github.com/ltlfuzzer/LTL-Fuzzer**

2 APPROACH OVERVIEW

At a high level, our approach takes a sequential program P and a Linear-time Temporal Logic (LTL) property ϕ as inputs. The atomic propositions in ϕ refer to predicates over the program variables that can be evaluated to true or false. An example is a predicate $x > y$ in which x and y are program variables. Our approach identifies program locations at which the atomic propositions in the LTL property may be affected. For this, we find program locations at which the values of variables in the atomic proposition and their aliases may change.[1] Our technique outputs a *counterexample*, i.e., a concrete program input that leads to a violation of the specification. Counterexample generation proceeds in two phases. In the first phase, the program P is transformed into P'. For this, we use code instrumentation to monitor program behaviors and state transitions during program execution. We check these against the provided LTL property. In the second phase, a fuzz campaign is launched for the program P' to find a counterexample through directed fuzzing.

We illustrate our technique with an FTP implementation called Pure-FTPd[2]. Pure-FTPd is a widely-used open source FTP server which complies with the FTP RFC[3]. Here is a property described in the RFC that an FTP implementation must satisfy. The FTP server must stop receiving data from a client and reply with code 552 when user quota is exceeded while receiving data. Code 552 indicates the allocated storage is exceeded. Throughout this paper, we will use this FTP property – as represented by ϕ – to illustrate how our technique finds property violations in Pure-FTPd.

2.1 LTL Property Construction

We start by manually translating the informal property in the RFC into a LTL property ϕ. For this, we search the Pure-FTPd source code using keywords `APPE` and `552`. Source code analysis reveals that (1) Pure-FTPd implements a quota-based mechanism to manage user storage space and it works only when activated, and (2) the command `APPE` is handled by the function `dostor()`, in which `user_quota_size` is checked when receiving data. When the quota is exceeded, the server replies with code 552 (`MSG_QUOTA_EXCEEDED`) via the function `addreply()`. We therefore construct the property ϕ as

$$\neg F(a \wedge F(o \wedge G\neg n)) \tag{1}$$

The negation of ϕ is thus

$$F(a \wedge F(o \wedge G\neg n))$$

where definition of atomic propositions a, o, n appear in Table 1.

Next, we identify program locations where the values of variables in atomic propositions in ϕ may change at runtime. A simple

[1] In general, our approach requires an alias analysis to map the atomic propositions to program locations.
[2] https://www.pureftpd.org/project/pure-ftpd/
[3] https://www.w3.org/Protocols/rfc959/

Table 1: Mapping between atomic propositions and program locations ("..." indicates omitted loop entries).

Predicate	Atomic Prop.	Locations
$quota\_activated = true$	a	$\langle ftpd.c, 6072\rangle$
$user\_dir\_size > user\_quota$	o	$\langle safe\_rw.c, 12\rangle$ $\langle safe\_rw.c, 43\rangle$
$msg\_quota\_exceeded = true$	n	$\langle ftpd.c, 4444\rangle$ $\langle ftpd.c, 3481\rangle$
$loop\_entry = true$	l	$\langle ftpd.c, 4067\rangle$ ...

example is the proposition $quota\_activated = true$, which corresponds to the program location where quota checking is enabled in Pure-FTPd. At another statement, $user\_dir\_size > user\_quota$, we consider the first statement of functions that are used to store data in user directories. As a result, whenever data is written to user directories, those functions will be invoked and this proposition will be evaluated, i.e., all cases where user quota is exceeded will be captured in an execution. For $msg\_quota\_exceeded = true$, we identify function invocations of `addreply(552, MSG_QUOTA_EXCEEDED...)` which are a reply to clients when the quota is exceeded. Specific program locations for each atomic proposition are listed in Table 1. Their corresponding code snippets are shown in Listings 1, 2, 3, and 4. Here, we show one code snippet per atomic proposition. For convenience, we use a tuple $\langle l, p, c_p\rangle$ in which l denotes a program location, p is an atomic proposition, and c_p represents the predicate for the atomic proposition p. At the end of our manual LTL property generation process, we output a list L comprising such tuples. For the example property, the manual process of writing down the predicates and the accompanying tuples was completed by one of the authors in 20 minutes.

Listing 1: Enabling the user quota option:<ftpd.c, 6072>.

```
6063   #ifdef QUOTAS
6064   case 'n': {
...
6072     user_quota_size *= (1024ULL * 1024ULL);
6073 +   if(1){
6074 +     generate_event("a");
6075 +     if(liveness) record_state();
6076 +   }
```

Listing 2: Writing to user directories:<safe_rw.c, 12>.

```
12  safe_write(const int fd, const void * const buf_,
13        size_t count, const int timeout)
14  {
15 +  if(user_dir_size > user_quota){
16 +    generate_event("o");
17 +    if(liveness) record_state();
18 +  }
```

Listing 3: Replying msg_quota_exceeded:<ftpd.c, 4444>.

```
4442   afterquota:
4443     if (overflow > 0) {
4444       addreply(552, MSG_QUOTA_EXCEEDED, name);
4445 +     if(1){
4446 +       generate_event("n");
4447 +       if(liveness) record_state();
4448 +   }
```

Listing 4: Entry of a loop statement:<ftpd.c. 4067>.

```
4066   for (;;) {
4067 +   if(1){
4068 +       generate_event("l");
4069 +       if(liveness) record_state();
4070 +   }
```

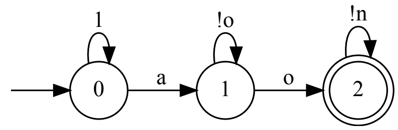

Figure 1: Büchi automata accepting traces satisfying $\neg\phi$.

2.2 Program Transformation

After deriving the property ϕ and the list of tuples L, we transform program P into P', which can report a failure at runtime whenever ϕ is violated. We perform this program transformation using two instrumentation modules: (1) *Event generator,* which generates an event when a proposition in ϕ is evaluated to true at runtime; (2) *Monitor,* which collects the generated events into an execution trace and evaluates if the trace violates ϕ. If a violation is found, the monitor reports a failure.

Event Generator. To detect changes in ϕ's proposition values during program execution, the event generator injects event generation statements at specific program locations. To do so, the generator takes the list L produced in the previous step as input. For each tuple $\langle l, p, c_p\rangle \in L$, the generator injects a statement `if(`c_p`) generate_event("`p`");` at the program location l, such that an event associated with p can be generated when condition c_p is satisfied. For instance, the program location $\langle ftpd.c, 6072\rangle$ corresponds to the proposition variable a ($quota\_activated = true$) and the enabling condition is `true`. The generator then inserts a statement `if(1) generate_event("a");` at line 6072 in ftpd.c (see Listing 1). Consequently, whenever $\langle ftpd.c, 6072\rangle$ is reached, an event associated with a is generated and recorded at runtime. Instrumentation for the other tuples appear in Listings 2, 3, and 4.

Monitor. The monitor module inserts a *monitor* into program P to verify if the program behavior conforms to property ϕ at runtime. Specifically, the monitor produces a trace τ by collecting events that are generated during execution (by the instrumented code). It then converts the negation of ϕ to a Büchi automata $\mathcal{A}_{\neg\phi}$, and checks whether $\mathcal{A}_{\neg\phi}$ accepts τ. If the trace is accepted, the monitor reports a failure, i.e., ϕ does not hold in P. In our Pure-FTPd example, the negation of ϕ is $F(a \wedge F(o \wedge G\neg n))$, and the converted Büchi automata $\mathcal{A}_{\neg\phi}$ is illustrated in Figure 1.

Checking Safety Properties. A Büchi automata accepts a trace τ if and only if τ visits an accepting state of the automata "infinitely often" (e.g., state 2 in Figure 1). For the negation of a *safety* property ($\neg\phi$), the Büchi automata $\mathcal{A}_{\neg\phi}$ accepts all traces which reach an accepting state, since all traces reaching an accepting state will loop there infinitely often. Since only a finite prefix of the trace is relevant for obtaining the counter-example of a safety property, the monitor thus outputs a counterexample if it witnesses a trace that leads to an accepting state in the Büchi automata $\mathcal{A}_{\neg\phi}$.

Checking Liveness Properties. The Büchi automata of the negation of ϕ accepts a trace τ if and only if τ visits an accepting state of $\mathcal{A}_{\neg\phi}$ "infinitely often" (e.g., state 2 in Figure 1). For instance, an infinite trace $a, o, (v)^{\omega}$ in which $v \neq n$ will be accepted by $\mathcal{A}_{\neg\phi}$. Formally, such a trace has the form $\tau = \tau_1(\tau_2)^{\omega}$ ($|\tau_2| \neq 0$), where τ_1 starts in an initial state of the Büchi automata $\mathcal{A}_{\neg\phi}$ and runs until an accepting state s of $\mathcal{A}_{\neg\phi}$, and τ_2 runs from the accepting state s back to itself. Witnessing a trace $\tau = \tau_1(\tau_2)^{\omega}$ in which τ_2 occurs "infinitely many times" is difficult in practice, since a fuzz campaign visits program executions which are necessarily of finite length. A straightforward approach to tackle this difficulty is to detect a loop in the trace and terminate execution when witnessing the loop occurs m times, e.g., $\tau = \tau_1, \overbrace{\tau_2\tau_2\cdots}^{m}$. This approach is insufficient because witnessing τ_2 for m times does not guarantee τ_2 occurs infinitely often, for instance `for (i=0; i<m+2; i++){... τ2...}` may generate τ_2 for m times but stops generating τ_2 after `i==m+1`.

In this paper, we record program states when events associated with atomic propositions occur in the execution and detect a state loop in the witnessed trace. If the execution of the state loop produces τ_2, that means, trace τ_2 can be generated infinitely many times by repeatedly going through the state loop. As a result, we assume that the witnessed trace can be extended to an infinite $\tau_1(\tau_2)^{\omega}$ shaped trace. Consider following two sequences witnessed in the execution

$$\tau_e = e_0 e_1 \cdots e_i e_{i+1} \cdots e_{i+h} e_i \cdots e_{i+h}$$

$$\tau_s = s_0 s_1 \cdots \overbrace{s_i s_{i+1} \cdots s_{i+h}}^{loop\ body} s_{i+h+1} \cdots s_{i+2h}$$

where τ_e is a sequence of events associated with atomic propositions that occur in the execution and τ_s is a sequence of program states that are recorded when events occur, for instance s_i indicates the program state that is recorded when the event e_i occurs. Suppose s_i is identical to s_{i+h+1}, then $s_i \cdots s_{i+h+1}$ is a state loop and its loop body is $s_i \cdots s_{i+h}$. Whenever s_i takes input $I_{s_i \cdots s_{i+h+1}}$ that leads to s_i from s_{i+h+1}, s_i will transition to s_i itself. We assume that the system under test is a reactive system taking a sequence of inputs and it is deterministic, that is, the same input always leads to the same program behavior in the execution. Thus, $e_i e_{i+1} \cdots e_{i+h}$ can be generated infinitely many times by repeatedly executing input $I_{s_i \cdots s_{i+h+1}}$ on state s_i. Trace $\tau_e = e_0 \cdots e_{i-1}(e_i \cdots e_{i+h})^{\omega}$ can be generated by running input $I_{s_0 \cdots s_i}(I_{s_i \cdots s_{i+h+1}})^{\omega}$, where $I_{s_0 \cdots s_i}$ is an input that leads to state s_i from s_0 and $I_{s_i \cdots s_{i+h+1}}$ is an input that leads to s_i from s_{i+h+1}.

As explained, occurrence of a state loop in the execution is evidence that the witnessed trace can be extended to an infinite $\tau_1(\tau_2)^{\omega}$ shaped trace. We leverage this idea to find a violation of a liveness property. When witnessing a trace in the execution that can be extended to a $\tau_1(\tau_2)^{\omega}$ shaped trace that is accepted by Büchi automata $\mathcal{A}_{\neg\phi}$, we consider a violation of the liveness property has been found. Hence, for liveness property guided fuzzing, we enrich the program transformation of P to P' as follows: (1) instrumenting a function call that records the current program state when an event appearing in a transtion label of $\mathcal{A}_{\neg\phi}$ occurs in the execution (shown in Listing 1-4) — specifically, function call `record_state()` takes the current program state and generates a hash code for the state at runtime; (2) instrumenting event-generating and state-recording statements at the entries of `for` and `while` loop statements in the program to observe possible loops in fuzzing. Listing 4 shows the instrumentation of a `for` loop statement in Pure-FTPd. More detailed and specific optimizations about state saving for checking liveness properties, appear in Section 5.

2.3 Witnessing Event Sequences

Since program P', generated in the previous step, reports a failure when ϕ is violated, we can find a counterexample for ϕ by fuzzing P'. An input that leads to such a failure is a counterexample. However, finding an input of this kind is challenging because it has to generate an execution in which certain events occur in a specific order. In our running example of Pure-FTPd, the quota mechanism must be activated first in the execution, then *user_quota* must be exceeded, and finally the execution must enter a loop in which no *msg_quota_exceeded* is sent back to the client. Existing directed fuzzing approaches like AFLGo [17] aim to direct fuzzing towards a particular program location and cannot drive execution through multiple program locations in a specific order. We now discuss our Büchi automata guided fuzzing in the next section.

3 BÜCHI AUTOMATA GUIDED FUZZING

Given an LTL property ϕ to be checked, automata-theoretic model checking of LTL properties [62] constructs the Büchi automata $\mathcal{A}_{\neg\phi}$ accepting all traces satisfying $\neg\phi$. In this section we will discuss how $\mathcal{A}_{\neg\phi}$ can be used to guide fuzzing. First we design a mechanism to generate an input whose execution passes through multiple program locations in a specific order. We design this mechanism by augmenting a greybox fuzzer in two ways.

- Power scheduling. During fuzzing, the power scheduling component tends to select seeds *closer* to the target on the pre-built inter-procedural control flow graph. Thus, the target can be reached efficiently. To achieve this, we use the fuzzing algorithm of AFLGo [17].
- Input prefix saving. This component observes execution and records input elements that have been consumed when reaching a target.

As mentioned, we focus on fuzzing reactive systems that take a sequence of inputs. The mechanism we follow involves directing fuzzing towards multiple program locations in a specific order. Consider a sequence of program locations $l_1, l_2 \cdots l_m$. Our approach works as follows: first, it takes l_1 as the first target and focuses on generating an input that leads to l_1. Meanwhile, it observes execution and records the prefix i_1 that leads to l_1. Next, it takes l_2 as the target, and focuses on exploring the space of inputs starting with prefix i_1, i.e., keeping generating inputs starting with i_1. As a result, an input that reaches l_2 via l_1 can be generated.

Based on the above mechanism of visiting a sequence of program locations, we develop an automata-guided fuzzing approach. The approach uses the Büchi automata $\mathcal{A}_{\neg\phi}$ instrumented in program P' and observes the progress that each trace makes on $\mathcal{A}_{\neg\phi}$ at runtime, e.g., how many state transitions are made towards the accepting state. To guide fuzzing, the approach saves the progress each input achieves on $\mathcal{A}_{\neg\phi}$ and uses it to generate inputs that make further progress. Specifically, it saves the progress for each input by recording state transitions that are executed on $\mathcal{A}_{\neg\phi}$ and the input prefix that leads to those transitions. Consider input i_0 and its trace τ_0 goes from initial state s_0 to state s_m on automata $\mathcal{A}_{\neg\phi}$. The achieved progress is represented as a tuple $\langle x_0^i, x_0^s \rangle$, where x_0^i is the shortest prefix of i_0 whose execution trace goes from s_0 to s_m and x_0^s is the state transition sequence $s_0 \cdots s_m$ visited. Such *progress* tuples are stored into a set $\mathcal{X}$ and are used to guide fuzzing.

For input generation, the approach takes a tuple from $\mathcal{X}$ and uses it to generate inputs that makes further progress. Consider a tuple $\langle x^i, x^s \rangle$: x^s records state transitions on automata $\mathcal{A}_{\neg\phi}$ which input prefix x^i has led to. Thus, we can query $\mathcal{A}_{\neg\phi}$ with x^s to find a transition that makes further progress, i.e., a state transition that gets closer to an accepting state of $\mathcal{A}_{\neg\phi}$. In the example, assuming x^s is state 0 in Figure 1, then the transition from state 0 to state 1 will be identified since state 1 is closer to the accepting state 2. Suppose t is the next progressive state transition of x^s, then we can further query $\mathcal{A}_{\neg\phi}$ to obtain atomic propositions that trigger transition t. Then, by querying the map between atomic propositions and program locations, we can identify program locations for those atomic propositions. In the example, atomic proposition a triggers transition from state 0 to state 1 and its corresponding program location is $\langle ftpd.c, 6072 \rangle$, as shown in Table 1.

From the above we can define criteria for an input to make further progress: (1) its execution has to follow the path that an input prefix x^i has gone through such that the generated trace can go through state transitions x^s; and, (2) subsequently the execution reaches one of program locations that are identified above to ensure the generated trace takes a step further in $\mathcal{A}_{\neg\phi}$.

To generate inputs of this kind, our mechanism for generating inputs that traverse a sequence of program locations in a specific order comes into play. Assume l_i is one of program locations identified above, for making further "progress" in $\mathcal{A}_{\neg\phi}$. The mechanism takes l_i as the target and keeps generating inputs that start with prefix x^i until generating an input that starts with prefix x^i and subsequently visits location l_i. This is how our approach uses tuples in $\mathcal{X}$ to generate inputs that make further "progress" towards an accepting state in the Büchi automata $\mathcal{A}_{\neg\phi}$. The detailed fuzzing algorithm is now presented.

4 FUZZING ALGORITHM

Algorithm 1 shows the workflow of our counterexample guided fuzzing. To find a counterexample, the algorithm guides fuzzing in two dimensions. First, it prioritizes the exploration of inputs whose execution traces are more likely to be accepted by $\mathcal{A}_{\neg\phi}$. Specifically, if the trace of the prefix of an input reaches a state that is closer to an accepting state on $\mathcal{A}_{\neg\phi}$, then its trace is more likely to be accepted. The algorithm selects input prefixes whose traces have been witnessed to get close to an accepting state and keeps generating inputs starting with them (shown in line 5 and line 10). Secondly, the algorithm focuses on generating inputs whose execution makes further progress on $\mathcal{A}_{\neg\phi}$. Given an input prefix, the algorithm finds a state transition t that helps us get closer to an accepting state in $\mathcal{A}_{\neg\phi}$, and finds the atomic propositions which enable t to be taken (line 6). For the atomic propositions enabling transition t, we identify the corresponding program locations (line 7). Then we attempt to generate inputs that reach the program location in the execution and trigger the program behavior associated with the atomic proposition. As a result, the generated trace can make further progress in $\mathcal{A}_{\neg\phi}$. To generate inputs that reach a particular program location, we leverage the algorithm proposed in AFLGo (line 8-14). Its idea is to assign more power to seeds that are *closer* to the target on a pre-built control flow graph such that the generated inputs are more likely to reach the target. The time budget for reaching a target is configurable, via parameter $target\_time$.

Algorithm 1: Counterexample-Guided Fuzzing

Input: P': The transformation of program under test
Input: $\mathcal{A}_{\neg\phi}$: Automata of negation of property under test
Input: map: Map between propositions and program locations
Input: $flag$: True for liveness properties
Input: $total\_time$: Time budget for fuzzing
Input: $target\_time$: Time budget for reaching a program location

1 **Procedure** Fuzz(P', $\mathcal{A}_{\neg\phi}$, map, $flag$, $total\_time$, $target\_time$)
2 $s_0 \leftarrow$ getInitState($\mathcal{A}_{\neg\phi}$) ;
3 $\mathcal{X} \leftarrow \{\langle \emptyset, s_0 \rangle\}$; // Starting with init state of $\mathcal{A}_{\neg\phi}$
4 **for** $time < total\_time$ **do**
5 $\langle x_t^i, x_t^s \rangle \leftarrow$ selectPrefix($\mathcal{X}$) ;
6 $p \leftarrow$ selectTargetAtomicProposition($\mathcal{A}_{\neg\phi}, x_t^s$) ;
7 $l \leftarrow$ selectProgramLocationTarget(map, p) ;
8 **for** $time' < target\_time$ **do**
// $\mathcal{D}$: Feedback of CFG distance
// S_{power}: Power schedule algorithm
9 $I \leftarrow$ generateInput($\mathcal{D}, S_{power}$) ;
10 $I' \leftarrow$ replacePrefix(I, x_t^i) ;
11 $d, \langle x^i, x^s \rangle \leftarrow$ evaluate($P', I', flag$);
12 $\mathcal{D} \leftarrow \mathcal{D} \cup \{d\}$;
13 $\mathcal{X} \leftarrow \mathcal{X} \cup \{\langle x^i, x^s \rangle\}$
14 **end**
15 **end**

For prefix selection (line 5), the algorithm defines a fitness function to compute a fitness value for each prefix tuple. Given a tuple $\langle x_t^i, x_t^s \rangle$, its fitness value is

$$f_t = \frac{l_s}{l_s + l_a} + \frac{1}{l_i}$$

where l_s is the length of x_t^s and l_a is the length of the shortest path from the last state of x_t^s to an accepting state on $\mathcal{A}_{\neg\phi}$ and l_i is the length of input prefix x_t^i. As shown in the formula, a prefix tuple has a higher fitness value if the last state of x_t^s is closer to an accepting state on $\mathcal{A}_{\neg\phi}$ and the input prefix is shorter. Heuristically, by extending such a prefix, our fuzzing algorithm is more likely to generate an input whose execution trace is accepted by $\mathcal{A}_{\neg\phi}$. Prefix tuples with higher fitness values are prioritized for selection.

For atomic proposition selection (line 6), we adopt a random selection strategy. Consider tuple $\langle x_t^i, x_t^s \rangle$ and the last state of x_t^s is s_t, the algorithm identifies atomic propositions that make a progressive transition from s_t on $\mathcal{A}_{\neg\phi}$ as follows: if state s_t is not an accepting state of $\mathcal{A}_{\neg\phi}$, any atomic proposition that triggers a transition from s_t towards an accepting state is selected. If state s_t is an accepting state, any atomic proposition that triggers a transition from s_t back to itself is selected. For simplicity, the algorithm randomly selects one from the identified atomic propositions. When the selected proposition p has multiple associated program locations, we randomly select one of them as a target. The main consideration for adopting a random strategy is to keep our technique as simple as possible. Moreover, these strategies can be configured in our tool.

5 STATE SAVING

In liveness property verification, LTL-Fuzzer detects a state loop in the witnessed trace. If a state loop is detected, LTL-Fuzzer assumes the current trace can be extended to a lasso-shaped trace $\tau_1(\tau_2)^\omega$. This works with a *concrete* representation of program states, however in reality state representation of software implementations are always abstracted. State representations that are too abstract may miss capturing variable states that are relevant to the loop, which leads to false positives. State representations that are too concrete may contain variable states that are irrelevant to the loop such as a variable for system-clock, which leads to false negatives. To be practical, LTL-Fuzzer takes a snapshot of application's registers and *addressable* memory and hashes it into a 32-bit integer, which is recorded as a state. Addressable memory indicates two kinds of objects: (1) global variables (2) objects that are explicitly allocated with functions `malloc()` and `alloca()`. Such a convention was also adopted in previous works on infinite loop detection [20, 59].

Furthermore, LTL-Fuzzer only records a program state for selected program locations, not for all program locations. Specifically, we only save states for the program locations associated with the transition labels of the automata $\mathcal{A}_{\neg\phi}$ where ϕ is the liveness property being checked. Note that a transition label in $\mathcal{A}_{\neg\phi}$ is a subset of atomic propositions [62, 63]. The full set of atomic propositions is constructed by taking the atomic propositions appearing in ϕ and embellishing this set with atomic propositions that we introduce for occurrence of each program loop header (such as l in Table 1). If the transition label involves a set L of atomic propositions, we track states for only those atomic propositions in L which correspond to loop header occurrences. The goal here is to quickly find possible infinite loops by looking for a loop header being visited with the same program state. Hence for the transition label $!n$ in our running example, we only store states for the program locations corresponding atomic proposition l in Table 1.

Listing 5: Quota checking:<ftpd.c. 4315>.

```
4315  if(...(max_filesize >= (off_t) 0 &&
            (max_filesize=user_quota_size - quota.size)
            < (off_t) 0 )){
        ...
4322      goto afterquota;
4323  }
```

In the example shown in Section 2, LTL-Fuzzer witnesses a state generated at program location $\langle ftpd.c, 4067\rangle$ (shown in Listing 4) that has been observed before and at the same time the witnessed trace is accepted by $\mathcal{A}_{\neg\phi}$. In this case, LTL-Fuzzer reports a violation of the LTL property ϕ shown in Page 2. To validate if the violation is spurious, we check if the observed state loop can be repeated in the execution. Our analysis shows a chunk of data was read during the execution of the state loop and the chunk of data was from a file uploaded by the client. We duplicated the chunk of data in the uploaded file and reran the experiment and found the state loop was repeated. That means the witnessed trace can be extended to a $\tau_1(\tau_2)^\omega$ shaped trace, which visits the accepting state of the automata accepting $\neg\phi$ (shown in Figure 1) infinitely many times. Thus, the reported violation is not spurious.

We further analyzed the root cause of the violation. It shows there was a logical bug in the quota checking module. As shown in Listing 5, the assignment of `max_filesize` occurs in a conditional statement and is never executed due to that `max_filesize`'s initial value is -1. To fix the bug, we created a patch and submitted a pull request on the Github repo of Pure-FTPd, which has been confirmed and verified.

6 LTL-FUZZER IMPLEMENTATION

We implement LTL-Fuzzer as an open source tool built on top of AFL, which comprises two main components: instrumentor and fuzzer. In the following, we explain these components.

6.1 Instrumentation Module

AFL comes with a special compiler pass for `clang` that instruments every branch instruction to enable coverage feedback. By extending this compiler, we instrument a program under test at three levels: specific locations, basic blocks, and the application.

Specific locations. LTL-Fuzzer takes a list of program locations at which program behaviors associated with a property under test might occur. At each of the given program locations, the instrumentation module injects two components: *event generator* and *state recorder*. Event generator is a piece of code that generates an event when the provided condition is satisfied at run-time. The state recorder is a component that takes a snapshot of program states and generates a hash code for the state when the given program location is reached in the execution.

Basic blocks. LTL-Fuzzer guides fuzzing to a target using the feedback on how close to the target an input is as explained in Section 3. At runtime, LTL-Fuzzer requires the distance from each basic block to the target on the CFG (control flow graph). The instrumentor instruments a function call in each basic block at runtime. The function call will query a table that stores distances from each block to program locations associated with the given property (i.e., targets). The distance from a basic block to each program location is computed offline with the distance calculator component that is borrowed from AFLGo [17].

Applications. For a program under test, the instrumentation module injects a *monitor* into the program. During fuzzing, the monitor collects events generated by instrumented event generators and produces execution traces. For property checking, the monitor leverages Spot libraries [10] to generate a Büchi automata from the negation of an LTL property and validates these traces. The instrumentation module also instruments an *observer* in the program that monitors execution of inputs; it maps a given suitable execution trace prefix to the input event sequence producing it, so that the occurrence of the prefix can be detected by the observer, during fuzzing. The fuzzing process then seeks to further extend this prefix with "suitable" events as described in the following.

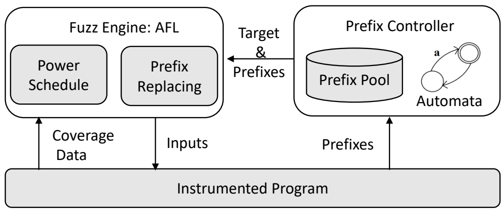

Figure 2: The architecture of LTL-Fuzzer.

6.2 Fuzzer

Figure 2 shows the Fuzzer component's architecture. It mainly comprises two modules: *prefix controller* and *fuzz engine*. LTL-Fuzzer saves input prefixes whose execution traces make transitions on the automata and reuses them for further exploration (Section 3). At runtime, the prefix controller conducts three tasks: (1) collecting prefixes reported by the monitor instrumented in the program under test and storing them into a pool; (2) selecting a prefix from the pool for further exploration according to Algorithm 1; (3) identifying the target program location based on the selected prefix. The fuzz engine is obtained by modifying AFL [1]. It generates inputs starting with a given input prefix. To reach a target, our fuzzer integrates the power scheduling component developed in AFLGo [17] to direct fuzzing. In LTL-Fuzzer, we direct execution to reach a target after the execution of an input prefix. Thus, the fuzz engine collects no feedback, such as coverage data during execution of the input prefix, and only collects feedback data after the execution of the input prefix is completed.

7 EVALUATION

In our experiments, we seek to answer the following questions:

RQ1 Effectiveness: How effective is LTL-Fuzzer at finding LTL property violations?

RQ2 Comparison: How does LTL-Fuzzer compare to the state-of-the-art validation tools in terms of finding LTL property violations?

RQ3 Usefulness: How useful is LTL-Fuzzer in revealing LTL property violations in real-world systems?

7.1 Subject Programs

Table 2 lists the subject programs used in our evaluation. This includes 7 open source software projects that implement 6 widely-used network protocols. We selected these projects because they (1) are reactive software systems that LTL-Fuzzer is designed for, (2) include appropriate specification documents from which LTL properties can be generated, and (3) are widely-used and have been studied. Finding bugs in such real-world systems is thus valuable.

7.2 Experiment Setup

To answer the research questions, we conducted three empirical studies on the subject programs.

7.2.1 Effectiveness of LTL-Fuzzer. We evaluate LTL-Fuzzer's effectiveness by running it on a set of LTL properties in subject programs where violations are already known; we check the number of LTL properties for which LTL-Fuzzer can find violations.

Table 2: Detailed information about our subject programs.

Project	Protocol	#SLOC	InPreviousWork	GithubStars
ProFTPD [7]	FTP	210.8k	[52]	339
Pure-FTPd [9]	FTP	52.9k	[52]	435
Live555 [4]	RTSP	52.5k	[53] [52]	526
OpenSSL [6]	TLS	286.7k	[40] [52] [27]	16.3K
OpenSSH [5]	SSH	98.3k	[32] [52]	1.5K
TinyDTLS [11]	DTLS	63.2k	[31] [52]	43
Contiki-Telnet [2]	TELNET	353.4k	[40]	3.4K

To create such a dataset, we collect event ordering related CVEs (so that they can be captured as a temporal property) that are disclosed in subject programs, e.g., an FTP client copies files from the server without logging in successfully. Specifically, for each subject, we select 10 such CVEs with criteria: (1) reported recently (during 2010-2020); (2) include instructions to reproduce the bug, (3) relevant to event orderings. Then we manually reproduce them with the corresponding version of code. If a CVE is reproducible, then we write the property in LTL and put it in our dataset of LTL properties. Based on the aforementioned criteria, we collected 14 CVEs in 7 subjects as shown in Table 3; these LTL properties can be found in our dataset [4] and the appendix of our arxiv paper [5]. Our goal is to check experimentally if LTL-Fuzzer can find violations of these LTL properties.

7.2.2 Comparison with other tools. We evaluate LTL-Fuzzer and state-of-the-art techniques on the LTL property dataset above and compare them in terms of the number of LTL properties for which each technique finds the violations and the time that is used to find a violation. For state-of-the-art techniques, we reviewed recent and well-known techniques in model checking, runtime verification and directed fuzzing domains. We chose the following techniques for comparison with LTL-Fuzzer.

- AFLGo [17]. It is a well known directed greybox fuzzer which drives execution to a target with a simulated annealing-based power schedule that assigns more energy to inputs that hold the trace closer to the target. We take it as a baseline tool.
- AFL_{LTL}. It is an implementation which enables AFLGo to detect an LTL property violation. Specifically, AFL_{LTL} powers AFLGo with only the LTL test oracle such that it can report an error when the given LTL property is violated in the execution. By comparing with AFL_{LTL}, we evaluate how effective is our automata-guided fuzzing strategy in finding LTL property violations. Note that AFL_{LTL} is also a tool built by us, but it lacks the automata guided fuzzing of LTL-Fuzzer.
- L+NuSMV. It combines model learning and model checking to verify properties in a software system. Specifically, it leverages a learning library called LearnLib [41] to build a model for the software system and then verifies given properties on the learned model with the well-known model checker NuSMV [23]. In the paper, we indicate it with L+NuSMV. This technique was published at CAV 2016 [30] and has been subsequently adopted in recent works such as [67] and [31].

[4] https://github.com/ltlfuzzer/LTL-Fuzzer/tree/main/ltl-property
[5] https://arxiv.org/abs/2109.02312

Table 3: Statistics of found violations and the performance of four tools in finding the violations.

Prop	CVE-ID	Type of Vulnerability	Program	Version	LTL-Fuzzer	AFL$_{LTL}$		AFLGo		L+NuSMV	
					Time(h)	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$
PrF_1	CVE-2019-18217	Infinite Loop	ProFTPD	1.3.6	4.62	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
PrF_2	CVE-2019-12815	Illegal File Copy	ProFTPD	1.3.5	0.95	2.01	**0.84**	T/O	**1.00**	T/O	**1.00**
PrF_3	CVE-2015-3306	Improper Access Control	ProFTPD	1.3.5	1.14	1.89	**0.76**	T/O	**1.00**	T/O	**1.00**
PrF_4	CVE-2010-3867	Illegal Path Traversal	ProFTPD	1.3.3	2.06	5.17	**0.85**	T/O	**1.00**	T/O	**1.00**
LV_1	CVE-2019-6256	Improper Condition Handle	Live555	2018.10.17	5.29	11.13	**1.00**	11.47	**1.00**	T/O	**1.00**
LV_2	CVE-2019-15232	Use after Free	Live555	2019.02.03	0.22	1.42	**0.91**	1.46	**0.92**	T/O	**1.00**
LV_3	CVE-2019-7314	Use after Free	Live555	2018.08.26	1.27	4.18	**0.98**	T/O	**1.00**	T/O	**1.00**
LV_4	CVE-2013-6934	Numeric Errors	Live555	2013.11.26	2.73	2.58	0.40	2.21	0.39	T/O	**1.00**
LV_5	CVE-2013-6933	Improper Operation Limit	Live555	2011.12.23	1.80	1.99	0.63	1.45	0.33	T/O	**1.00**
SH_1	CVE-2018-15473	User Enumeration	OpenSSH	7.7p1	0.18	0.17	0.44	T/O	**1.00**	24.00	**1.00**
SH_2	CVE-2016-6210	User Information Exposure	OpenSSH	7.2p2	0.19	0.19	0.50	T/O	**1.00**	24.00	**1.00**
SL_1	CVE-2016-6309	Use after Free	OpenSSL	1.1.0a	3.77	6.00	**0.74**	6.58	**0.82**	T/O	**1.00**
SL_2	CVE-2016-6305	Infinite Loop	OpenSSL	1.1.0	1.45	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
SL_3	CVE-2014-0160	Illegal Memory Access	OpenSSL	1.0.1f	1.11	7.31	**1.00**	T/O	**1.00**	T/O	**1.00**
Found violations in total			-		14	12		5		2	
Average time usage (hours)			-		1.91	6.57		17.08		24.00	
Comparison with LTL-Fuzzer on time usage			-		-	3.44x		8.93x		12.55x	

[1] T/O represents tools cannot expose vulnerabilities within 24 hours for 10 experimental runs. We replace T/O with 24 hours when calculating average usage time.
[2] Statistically significant values of $\hat{A}_{12}$ are shown in bold.

We briefly summarize why we did not include certain other model-checkers and fuzzers, and all runtime verification tools for comparison. Model checking tools CBMC [24], CPAChecker [16][6], Seahorn [36], SMACK [55], UAutomizer [38], DIVINE [14] cannot support LTL property verification. Schemmel's work [59] published at CAV 2018 partially supports LTL property verification. SPIN [39] supports LTL property verification but only works with a modeling language Promela [8] and the tool provided in SPIN for extracting models from C programs failed to work on our subject programs. Some model checking tools [40, 60], and directed fuzzing tools (like UAFL [66], Hawkeye [22] and TOFU [68]) we reviewed, are not publicly available.

Finally, all of available runtime verification tools [29] (like JavaMOP [42], MarQ [57] and Mufin [28]) cannot check LTL properties in C/C++ software systems. Furthermore, our method is conceptually different and complementary to runtime verification — our method generates test executions, while runtime verification checks a test execution. While the combination of our method with runtime verification is possible, a comparison is less meaningful.

7.2.3 Real-world utilty. In this study, we read RFC specifications that these subject programs follow to extract temporal properties and describe them in LTL. Then we use LTL-Fuzzer to check these properties on the subject programs.

Configuration Parameters. Following fuzzing evaluation suggestions from the community [46], we run each technique for 24 hours and repeat each experiment 10 times to achieve statistically significant results. For the initial seeds, we use seed inputs provided in ProFuzzBench [52] for all subjects. ProFuzzBench is a benchmark for stateful fuzzing of network protocols, which contains a suite of representative open-source network protocol implementations. For Contiki-Telnet, which is not contained in ProFuzzBench, we generate random inputs as its initial seeds. For LTL-Fuzzer, we need to specify the time budget for reaching a single program location and we configure it with 45 minutes for each target. For AFLGo and AFL$_{LTL}$, we need to provide a target for an LTL property being checked. We specify the target by randomly selecting from program locations that are associated with atomic propositions that trigger the transition to an accepting state on the automata of the negation of the property. In the example in Section 2, we chose one of loop entries as the target since proposition o triggers the transition to the accepting state shown in Figure 1 and it corresponds with loop entries. For execution environments, we conducted experiments on a physical machine with 64 GB RAM and a 56 cores Intel(R) Xeon(R) E5-2660 v4 CPU, running a 64-bit Ubuntu TLS 18.04 as the operating system.

7.3 Experimental Results

7.3.1 [RQ1] Effectiveness. Table 3 shows property violations found by LTL-Fuzzer for the 14 LTL properties derived from known CVEs. The first column shows identifiers of the properties being checked. The corresponding LTL properties and their descriptions can be found in our dataset. Columns 2 - 5 represent CVE-IDs, types of vulnerabilities that CVEs represent, subject names, and subject versions, respectively. Column "LTL-Fuzzer" shows the time that is used to find a violation by LTL-Fuzzer. As shown in Table 3, LTL-Fuzzer can effectively detect violations of LTL properties in the subjects. It successfully detected the violation for all the 14 LTL properties in the dataset. On average, it took LTL-Fuzzer 1.91 hours to find a violation.

[6]For some tools, the LTL checker module is not available for usage / experimentation, as our email enquiry with CPAchecker team revealed.

Table 4: Zero-day Bugs found by LTL-Fuzzer; for several of them CVEs have been assigned but CVE ids are not shown.

Prop	Project	Description of violated properties	Bug Status
TD_1	TinyDTLS (0.9-rc1)	If the server is in the `WAIT_CLIENTHELLO` state and receives a `ClientHello` request with valid cookie and the epoch value 0, must finally give `ServerHello` responses.	CVE-2021-42143, Fixed
TD_2	TinyDTLS (0.9-rc1)	If the server is in `WAIT_CLIENTHELLO` state and receives a `ClientHello` request with valid cookie but not 0 epoch value, must not give `ServerHello` responses before receiving `ClientHello` with 0 epoch value.	CVE-2021-42142, Fixed
TD_3	TinyDTLS (0.9-rc1)	If the server is in the `WAIT_CLIENTHELLO` state and receives a `ClientHello` request with an invalid cookie, must reply `HelloVerifyRequest`.	CVE-2021-42147, Fixed
TD_5	TinyDTLS (0.9-rc1)	If the server is in the `DTLS_HT_CERTIFICATE_REQUEST` state and receives a `Certificate` request, must give a `DTLS_ALERT_HANDSHAKE_FAILURE` or `DTLS_ALERT_DECODE_ERROR` response, or set `Client_Auth` to be verified.	CVE-2021-42145, Fixed
TD_{11}	TinyDTLS (0.9-rc1)	After the server receives a `ClientHello` request without renegotiation extension and gives a `ServerHello` response, then receives a `ClientHello` again, must refuse the renegotiation with an Alert.	Confirmed
TD_{12}	TinyDTLS (0.9-rc1)	After the server receives a `ClientHello` request and gives a `ServerHello` response, then receives a `ClientKeyExchange` request with a different epoch value than that of `ClientHello`, server must not give `ChangeCipherSpec` responses.	CVE-2021-42141, Fixed
TD_{13}	TinyDTLS (0.9-rc1)	After the server receives a `ClientHello` request and gives a `ServerHello` response, then receives a `ClientHello` request with the same epoch value as that of the first one, server must not give `ServerHello`.	CVE-2021-42146
TD_{14}	TinyDTLS (0.9-rc1)	If the server receives a `ClientHello` request and gives a `HelloVerifyRequest` response, and then receives a over-large packet even with valid cookies, the server must refuse it with an `Alert`.	CVE-2021-42144, Fixed
CT_1	Contiki-Telnet (3.0)	After `WILL` request is received and the corresponding option is disabled, must send `DO` or `DONT` responses.	CVE-2021-40523
CT_2	Contiki-Telnet (3.0)	After `DO` request is received and the corresponding option is disabled, must send `WILL` or `WONT` responses.	Confirmed
CT_7	Contiki-Telnet (3.0)	After `WONT` request is received and the corresponding option is disabled, must not give responses.	CVE-2021-38311
CT_8	Contiki-Telnet (3.0)	After `DONT` request is received and the corresponding option is disabled, must not give responses.	Confirmed
CT_{10}	Contiki-Telnet (3.0)	Before `Disconnection`, must send an Alert to disconnect with clients.	CVE-2021-38387
CT_{11}	Contiki-Telnet (3.0)	If conducting `COMMAND` without `AbortOutput`, the response must be same as the real execution results.	CVE-2021-38386
PuF_5	Pure-FTPd (1.0.49)	When quota mechanism is activated and user quota is exceeded, must finally reply a quota exceed message.	CVE-2021-40524, Fixed

> LTL-Fuzzer is found to be effective in finding LTL property violations, detecting violations for all 14 properties derived from known CVEs.

7.3.2 [RQ2] Comparison. As shown in Table 3, the last three main columns show the time that is used for comparison techniques to find a violation on the 14 LTL properties in the experiment. Note that "T/O" indicates a technique failed to find the violation for an LTL property in the given time budget (i.e., 24 hours). To mitigate randomness in fuzzing, we adopted the Vargha-Delaney statistic $\hat{A}_{12}$ [64] to evaluate whether one tool significantly outperforms another in terms of the time that is used to find a violation. The $\hat{A}_{12}$ is a non-parametric measure of effect size and gives the probability that a randomly chosen value from data group 1 is higher or lower than one from data group 2. It is commonly used to evaluate whether the difference between two groups of data is significant. Moreover, we also use Mann-Whitney U test to measure the statistical significance of performance gain. When it is significant (taking 0.05 as a significance level), we mark the $\hat{A}_{12}$ values in bold.

LTL-Fuzzer found violations of all of the 14 LTL properties, followed by AFL_{LTL} (12), AFLGo (5), and L+NuSMV (2). We note that AFL_{LTL} is also a tool built by us, it partially embodies the ideas in LTL-Fuzzer and is meant to help us understand the benefits of automata-guided fuzzing. In terms of the time that is used to find a violation, LTL-Fuzzer is the fastest (1.91 hours), followed by AFL_{LTL} (6.57 hours), AFLGo (17.08 hours), and L+NuSMV (24.00 hours). In other words, LTL-Fuzzer is 3.44x, 8.93x, 12.55x faster than AFL_{LTL}, AFLGo, and L+NuSMV, respectively. For CVE-2013-6934 and CVE-2013-6933, AFLGo performed slightly better than other techniques, while AFL_{LTL} exhibited the same performance as LTL-Fuzzer for CVE-2018-15473 and CVE-2016-6210. We investigated these 4 CVEs and found that triggering those vulnerabilities is relatively straightforward. They can be triggered without sophisticated directing strategies. As a result, other techniques achieve a slightly better performance than LTL-Fuzzer for these four CVEs. In terms of the $\hat{A}_{12}$ statistic, LTL-Fuzzer performs significantly better than other techniques in most cases.

> LTL-Fuzzer found violations of all the 14 LTL properties in the experiment. AFL_{LTL}, AFLGo and L+NuSMV found 12, 5, 2 property violations, respectively. LTL-Fuzzer is 3.44x, 8.93x, 12.55x faster than AFL_{LTL}, AFLGo, and L+NuSMV.

7.3.3 [RQ3] Real-world utility. In this study, we evaluate utility of LTL-Fuzzer by checking whether it can find zero-day bugs in real-world protocol implementations. We extract 50 properties from RFCs that our subject programs follow (aided by comments in the source code of the programs) and write them in linear-time temporal logic. The details of the 50 LTL properties can be found in our dataset. In the experiment, LTL-Fuzzer achieved a promising result. Out of these 50 LTL properties, LTL-Fuzzer discovered *new* violations for 15 properties, which are shown in Table 4. We reported these 15 zero-day bugs to developers and all of them got confirmed by developers. We reported them on the common vulnerabilities and exposures (CVE) system (see https://cve.mitre.org/) and 12 of them were assigned CVE IDs. Out of 15 reported violations, 7 have been fixed at the time of the submission of our paper. Notably, LTL-Fuzzer shows effectiveness in finding violations for liveness properties. In the experiment, LTL-Fuzzer successfully found violations for 4 liveness properties which are PrF_1, SL_2, TD_1 and PuF_5. All the 4 violations were confirmed by developers, i.e., they are not spurious results. Moreover, to discover violations for these 4 liveness properties, LTL-Fuzzer only recorded 6, 11, 4 and 9 states, respectively. Since every state is recorded as a 32-bit integer, the memory consumption for recording states is thus found to be negligible in our experiments.

Among 50 LTL properties extracted from protocol RFCs, LTL-Fuzzer found 15 previously unknown violations in protocol implementations and 12 of these have been assigned CVEs.

7.4 Threats to validity

There are potential threats to validity of our experimental results. One concern is *external validity*, i.e., the degree to which our results can be generalized to and across other subjects. To mitigate this concern, we selected protocol implementations that are widely used and have been frequently evaluated in previous research (as shown in Table 2). We may have made mistakes in converting informal requirements into LTL properties. To reduce this kind of bias, we let two authors check generated properties and remove those on which they do not agree, or do not think are important properties.

In principle, LTL-Fuzzer can report false positives due to incorrect instrumentation, e.g., if we fail to instrument some target locations for an atomic proposition. We mitigate the risk of false positives by checking the reported counterexamples and validating that they are true violations of the temporal property being checked. We add here that we did not encounter such false positives in any of our experiments.

Another concern is *internal validity*, i.e., the degree to which our results minimize systematic error. First, to mitigate spurious observations due to the randomness in the fuzzers and to gain statistical significance, we repeated each experiment 10 times and reported the Vargha-Delaney statistic $\hat{A}_{12}$. Secondly, our LTL-Fuzzer implementation may contain errors. To facilitate scrutiny, we make LTL-Fuzzer code available.

8 RELATED WORK

Model Checkers. Model checking is a well-known property verification technique dating back to 1980s [25, 54]; it is used to prove a temporal property in a finite state system, or to find property violation bugs. The early works check a temporal logic property against a finite state transition system. There exist well-known model checkers such as [23, 39, 43] which can be used to check temporal properties on a constructed model (via state space exploration). To construct models, one method is manual construction via a modeling language. This requires substantial effort and can be error-prone [35, 50]. LTL-Fuzzer directly checks software implementations; it does not separately extract models from software.

Early works on model checking have been extended to automatically find bugs in software systems, which are typically infinite-state systems. Model checking of software systems usually involves either some extraction of finite state models, or directly analyzing the infinite state software system via techniques such as symbolic analysis. Automatic model extraction approaches [12, 26, 37, 58] include the works on predicate abstraction and abstraction refinement [12, 13] which build up a hierarchy of finite-state abstract models for a software system for proving a property. These approaches extract models which are conservative approximations and capture a superset of the program behavior. There are a number of stateful software model checkers, such as CMC [50], Java Pathfinder [65], MaceMC [44], CBMC [24], CPAchecker [16], which find assertion violations in software implementations. Many of these checkers do not check arbitrary LTL properties for software implementations. These model checkers either suffer from state space explosion, or suffer from other kinds of explosion such as the explosion in the size/solving-time for the logical formula in bounded model checking. In contrast, LTL-Fuzzer does not save any states for safety property checking and saves only certain property-relevant program states in liveness property checking. At the same time, LTL-Fuzzer does not give verification guarantees and does not perform complete exploration of the state space. We now proceed to discuss incomplete validation approaches.

Incomplete Checkers. Instead of exploring the complete set of behaviors, or a super-set of behaviors, one can also explore a subset of behaviors. Incomplete model learning approaches [61] can be mentioned in this regard. The active model learning technique, such as LearnLib [41], is widely used to learn models of real-world protocol implementations [27, 30–32]. It does not need user involvement. But it is time-consuming and hard to determine whether the learned model represents the complete behavior of the software system [61, 69]. Compared with the active learning, LTL-Fuzzer can more rapidly check properties, as shown in our experimental comparison with LearnLib+NuSMV. To alleviate the state-explosion problem, stateless checkers such as VeriSoft [33] and Chess [51] have been proposed; these checkers do not store program states. These works typically involve specific search strategies to check *specific* classes of properties such as deadlocks, assertions and so on. In contrast, LTL-Fuzzer represents a general approach to find violations of *any* LTL property.

Runtime Verification. Runtime verification is a lightweight and yet rigorous verification technique [15, 48]. It analyzes a single execution trace of a system against formally specified properties (e.g., LTL properties). It origins from model checking and applies model checking directly to the real implementations. Model checking checks a model of a target system to verify correctness of the system, while runtime verification directly checks the implementation, which could avoid different behaviours between models and implementations. LTL-Fuzzer shares the same benefit as runtime verification. Besides, runtime verification deals with finite executions, as one single execution has necessarily to be finite. This avoids the state explosion problem that model checking suffers from. Meanwhile, it leads to that runtime verification approaches [21, 28, 42, 57] often only check safety properties. LTL-Fuzzer, however, is able to check liveness properties by leveraging the strategy of saving program states.

Conceptually, our method is very different from runtime verification. Runtime verification focuses on the checking (a temporal logic property) on a single execution. Our method is focused on using temporal logic property to guide the construction of an execution which violates the property. Thus our work is more of a test generation method. Since runtime verification methods need tests whose execution will be checked, our method can be complementary to runtime verification. In other words, our method can generate tests likely to violate a given temporal property, and these tests can further validated by run-time verification.

Greybox Fuzzing. There are three broad variants of fuzzing: blackbox fuzzing [49], whitebox fuzzing or symbolic execution [19, 40, 59], and greybox fuzzing [1, 17, 18, 22, 56, 66]. We first discuss greybox fuzzing since they are the most widely used in industry today. In contrast to software model checking, blackbox/greybox fuzzing techniques represent a random/biased-random search over the domain of inputs for finding bugs or vulnerabilities in programs. Most greybox fuzzing techniques are used to detect memory issues (e.g., buffer overflow and use after free) that can produce observable behaviors (e.g., crashes). However, LTL-Fuzzer can not only witness simple properties like memory corruption, but also detect LTL property violations, for *any* given LTL property, however complex. Recent advances in greybox fuzzing use innovative objective functions for achieving different goals, such as [17] directs the search to specific program locations. The capabilities of LTL-Fuzzer go beyond visiting specific locations, and LTL-Fuzzer is used to witness specific event ordering constraints embodied by the negation of an arbitrary LTL property. PGFUZZ [45] is a greybox-fuzzing framework to find safety violations for robotic vehicles, but it is customized to be used on implementations of robotic vehicles. LTL-Fuzzer can be used to find violation of *any* LTL property for software from any application domain.

Symbolic Execution based Validation. Symbolic execution or whitebox fuzzing approaches are typically used to find violations of simple properties such as assertions [19, 34]. Recent whitebox fuzzing techniques do find violations of certain classes of properties. Schemmel's work [59] checks liveness properties while CHIRON [40] checks safety properties. [70, 71] proposed regular-property guided dynamic symbolic execution to find the program paths satisfying a property. However, all of these approaches require a long time budget for heavy-weight program analysis and back-end constraint solving. As a result, these techniques face challenges in scalability. In contrast, LTL-Fuzzer is built on top of greybox fuzzing; it can validate arbitrarily large and complex software implementations.

9 PERSPECTIVE

We present LTL-Fuzzer, a linear-time temporal logic guided greybox fuzzing technique, which takes Linear-time Temporal Logic (LTL) properties extracted from informal requirements such as RFCs and finds violations of these properties in C/C++ software implementations. Our evaluation shows that LTL-Fuzzer is effective in finding property violations. It detected 15 LTL property violations in real world protocol implementations that were previously unknown; 12 of these zero day bugs have been assigned CVEs. We make the data-set of LTL properties, and our tool available for scrutiny.

Our work shows the promise of synergising concepts from temporal property checking with recent advances in greybox fuzzing (these advances have made greybox fuzzing more systematic and effective). Specifically, in this paper we have taken concepts from automata-theoretic model checking of LTL properties [62], while at the same time adapting/ augmenting directed greybox fuzzing [17]. The main advancement of greybox fuzzing in our work, is the ability to find violations of *arbitrary* LTL properties, which is achieved by borrowing the Büchi automata construction from [62]. We note that the real-life practical value addition of software model checking is often from automated bug-finding in software implementations rather than from formal verification. Runtime verification complements software model checking by analyzing a single execution trace of software implementations. Our work essentially shows the promise of enjoying the main practical benefits of software model checking more efficiently and effectively via augmentation of (directed) greybox fuzzing. This is partially shown by the experiments in this paper where we have compared our work with both model checkers and fuzzers. Our work is also complementary to runtime verification since we generate test executions guided by a LTL property, while runtime verification would check a LTL property against a single test execution.

Arguably we could compare LTL-Fuzzer with more model checkers and fuzzers, experimentally. At the same time, we have noted that many model checkers were found to be not applicable for checking arbitrary LTL properties of arbitrary C/C++ software implementations. Moreover, the problem addressed by LTL-Fuzzer is certainly beyond the reach of fuzzers since fuzzers cannot detect temporal property violations. Overall, we believe our work represents a practical advance over model checkers and runtime verification, and a conceptual advance over greybox fuzzers. We expect that the research community will take the work in our paper forward, to further understand the synergies among software model checking, runtime verification and greybox fuzzing.

REFERENCES

[1] August. 2021 (last accessed). AFL. https://lcamtuf.coredump.cx/afl/.
[2] August. 2021 (last accessed). Contiki-Telnet. https://github.com/contiki-os/contiki.
[3] August. 2021 (last accessed). libFuzzer. https://llvm.org/docs/LibFuzzer.html.
[4] August. 2021 (last accessed). Live555. http://www.live555.com/.
[5] August. 2021 (last accessed). OpenSSH. http://www.openssh.com/.
[6] August. 2021 (last accessed). OpenSSL. https://www.openssl.org/.
[7] August. 2021 (last accessed). ProFTPD. http://www.proftpd.org/.
[8] August. 2021 (last accessed). Promela. http://spinroot.com/spin/Man/promela.html.
[9] August. 2021 (last accessed). Pure-FTPd. https://www.pureftpd.org/project/pure-ftpd/.
[10] August. 2021 (last accessed). Spot. https://spot.lrde.epita.fr/.
[11] August. 2021 (last accessed). TinyDTLS. https://projects.eclipse.org/projects/iot.tinydtls.
[12] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic predicate abstraction of C programs. In *proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI 2001)*. ACM, 203–213. https://dl.acm.org/doi/abs/10.1145/378795.378846

[13] Thomas Ball and Sriram K. Rajamani. 2002. Automatically validating temporal safety properties of interfaces. In *proceedings of the 8th international SPIN workshop on Model checking of software (SPIN 2002)*. ACM, 103–122. https://dl.acm.org/doi/10.5555/380921.380932

[14] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. 2017. Model checking of C and C++ with DIVINE 4. In *proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA 2017)*. Springer, 201–207. https://link.springer.com/chapter/10.1007/978-3-319-68167-2_14

[15] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to runtime verification. In *Lectures on Runtime Verification*. Springer, 1–33. https://hal.inria.fr/hal-01762297

[16] Dirk Beyer and M Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *proceedings of the 23rd International Conference on Computer-Aided Verification (CAV 2011)*. Springer, 184–190. https://link.springer.com/chapter/10.1007/978-3-642-22110-1_16

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*. ACM, 2329–2344. https://dl.acm.org/doi/10.1145/3133956.3134020

[18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering (TSE)* 45, 5 (2017), 489–506. https://dl.acm.org/doi/10.1145/2976749.2978428

[19] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI 2008)*. USENIX Association, 209–224. https://dl.acm.org/doi/10.5555/1855741.1855756

[20] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and Escaping Infinite Loops with Jolt. In *proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP 2011)*. Springer, 609–633. https://doi.org/10.1007/978-3-642-22655-7_28

[21] Feng Chen and Grigore Roşu. 2007. Mop: an efficient and generic runtime verification framework. In *proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications (OOPSLA 2007)*. ACM, 569–588. https://doi.org/10.1145/1297027.1297069

[22] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Greybox Fuzzer. In *proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. ACM, 2095–2108. https://doi.org/10.1145/3243734.3243849

[23] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *proceedings of 14th International Conference on Computer-Aided Verification (CAV 2002)*. Springer, 359–364. https://link.springer.com/chapter/10.1007/3-540-45657-0_29

[24] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Springer, 168–176. https://link.springer.com/chapter/10.1007/978-3-540-24730-2_15

[25] Edmund M. Clarke and E Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. *Lecture Notes in Computer Science (LNCS)* 131 (1981), 55–71. https://link.springer.com/chapter/10.1007/BFb0025774

[26] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Pasareanu, Hongjun Zheng, et al. 2000. Bandera: Extracting finite-state models from Java source code. In *proceedings of the 22nd international conference on Software engineering (ICSE 2000)*. IEEE, 439–448. https://dl.acm.org/doi/10.1145/337180.337234

[27] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *proceedings of the 24th USENIX Conference on Security Symposium (USENIX Security 2015)*. USENIX Association, 193–206. https://dl.acm.org/doi/10.5555/2831143.2831156

[28] Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, and Daniel Thoma. 2016. Runtime monitoring with union-find structures. In *proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*. Springer, 868–884. https://doi.org/10.1007/978-3-662-49674-9_54

[29] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. 2021. A taxonomy for classifying runtime verification tools. *International Journal on Software Tools for Technology Transfer (STTT)* 23, 2 (2021), 255–284. https://hal.inria.fr/hal-01882410

[30] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *proceedings of 28th International Conference on Computer-Aided Verification (CAV 2016)*. Springer, 454–471. https://link.springer.com/chapter/10.1007/978-3-319-41540-6_25

[31] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *proceedings of the 29th USENIX Conference on Security Symposium Security (USENIX Security 2020)*. USENIX Association, 2523–2540. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[32] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. 2017. Model learning and model checking of SSH implementations. In *proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN 2017)*. ACM, 142–151. https://dl.acm.org/doi/10.1145/3092282.3092289

[33] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1997)*. ACM, 174–186. https://dl.acm.org/doi/10.1145/263699.263717

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2005)*. ACM, 213–223. https://dl.acm.org/doi/10.1145/1064978.1065036

[35] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In *proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*. ACM, 265–278. https://dl.acm.org/doi/10.1145/2043556.2043582

[36] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *proceedings of 27th International Conference on Computer-Aided Verification (CAV 2015)*. Springer, 343–361. https://link.springer.com/chapter/10.1007/978-3-319-21690-4_20

[37] John Hatcliff, Matthew B Dwyer, and Hongjun Zheng. 2000. Slicing software for model construction. *Higher-order and symbolic computation (LISP)* 13, 4 (2000), 315–353. https://dl.acm.org/doi/10.1023/A%3A1026599015809

[38] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software model checking for people who love automata. In *proceedings of 25th International Conference on Computer-Aided Verification (CAV 2013)*. Springer, 36–52. https://link.springer.com/chapter/10.1007/978-3-642-39799-8_2

[39] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering (TSE)* 23, 5 (1997), 279–295. https://dl.acm.org/doi/10.1109/32.588521

[40] Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. 2017. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017)*. IEEE, 627–638. https://ieeexplore.ieee.org/document/8023160

[41] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The open-source LearnLib. In *proceedings of the 27th International Conference on Computer-Aided Verification (CAV 2015)*. Springer, 487–495. https://link.springer.com/chapter/10.1007/978-3-319-21690-4_32

[42] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient parametric runtime monitoring framework. In *proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. IEEE, 1427–1430. https://ieeexplore.ieee.org/document/6227231

[43] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. 2015. LTSmin: high-performance language-independent model checking. In *proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*. Springer, 692–707. https://link.springer.com/chapter/10.1007/978-3-662-46681-0_61

[44] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, death, and the critical transition: Finding liveness bugs in systems code. In *proceedings of the 4th USENIX conference on Networked systems design and implementation (NSDI 2007)*. USENIX Association, 243–256. https://dl.acm.org/doi/10.5555/1973430.1973448

[45] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z Berkay Celik, and Dongyan Xu. 2021. PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. In *proceedings of 2021 Network and Distributed Systems Security Symposium (NDSS 2021)*. NDSS. https://dx.doi.org/10.14722/ndss.2021.24096

[46] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. ACM, 2123–2138. https://dl.acm.org/doi/10.1145/3243734.3243804

[47] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. IEEE, 602–613. https://ieeexplore.ieee.org/abstract/document/7582795

[48] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming (JLAP)* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[49] Barton P Miller, Gregory Cooksey, and Fredrick Moore. 2006. An empirical study of the robustness of macos applications using random testing. In *proceedings of the 1st international workshop on Random testing (RT 2006)*. ACM, 46–54. https://dl.acm.org/doi/10.1145/1145735.1145743

[50] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. 2002. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review (OSR)* 36, SI (2002), 75–88. https://dl.acm.org/doi/10.1145/844128.844136

[51] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. 2007. *Chess: A systematic testing tool for concurrent software.* Technical Report MSR-TR-2007-149. Microsoft Research. https://www.microsoft.com/en-us/research/publication/chess-a-systematic-testing-tool-for-concurrent-software/

[52] Roberto Natella and Van-Thuan Pham. 2021. ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing. In *proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. ACM, 662–665. https://dl.acm.org/doi/abs/10.1145/3460319.3469077?af=R

[53] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *proceedings of the 13th International Conference on Software Testing, Validation and Verification (ICST 2020)*. IEEE, 460–465. https://ieeexplore.ieee.org/document/9159093

[54] Jean-Pierre Queille and Joseph Sifakis. 1982. Specification and verification of concurrent systems in CESAR. In *proceedings of the 1982 International Symposium on programming*. Springer, 337–351. https://link.springer.com/chapter/10.1007/3-540-11494-7_22

[55] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling source language details from verifier implementations. In *proceedings of the 26th International Conference on Computer-Aided Verification (CAV 2014)*. Springer, 106–113. https://link.springer.com/chapter/10.1007/978-3-319-08867-9_7

[56] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *proceedings of 2017 Network and Distributed Systems Security Symposium (NDSS 2017)*, Vol. 17. NDSS, 1–14. http://dx.doi.org/10.14722/ndss.2017.23404

[57] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. 2015. MarQ: monitoring at runtime with QEA. In *proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*. Springer, 596–610. https://doi.org/10.1007/978-3-662-46681-0_55

[58] Ronald L Rivest and Robert E Schapire. 1993. Inference of finite automata using homing sequences. *Information and Computation (IANDC)* 103, 2 (1993), 299–347. https://dl.acm.org/doi/10.1145/73007.73047

[59] Daniel Schemmel, Julian Büning, Oscar Soria Dustmann, Thomas Noll, and Klaus Wehrle. 2018. Symbolic Liveness Analysis of Real-World Software. In *proceedings of 30th International Conference on Computer-Aided Verification (CAV 2018)*. Springer, 447–466. https://doi.org/10.1007/978-3-319-96142-2_27

[60] Markus Schordan, Jan Hückelheim, Pei-Hung Lin, and Harshitha Menon. 2017. Verifying the floating-point computation equivalence of manually and automatically differentiated code. In *proceedings of the 1st International Workshop on Software Correctness for HPC Applications (Correctness 2017)*. ACM, 34–41. https://dl.acm.org/doi/10.1145/3145344.3145489

[61] Frits Vaandrager. 2017. Model learning. *Communications of the ACM (CACM)* 60, 2 (2017), 86–95. https://dl.acm.org/doi/10.1145/2967606

[62] Moshe Y. Vardi and Pierre Wolper. 1986. An automata-theoretic approach to automatic program verification. In *proceedings of the 1st Symposium on Logic in Computer Science (LICS 1986)*. IEEE. https://orbi.uliege.be/handle/2268/116609

[63] Moshe Y. Vardi and Pierre Wolper. 1994. Reasoning about Infinite Computations. *Information and Computation (IANDC)* 115 (1994), 1–37. Issue 1. https://doi.org/10.1006/inco.1994.1092

[64] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics (JEBS)* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101

[65] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. Model checking programs. *Automated software engineering (ASE)* 10, 2 (2003), 203–232. https://link.springer.com/article/10.1023/A:1022920129859

[66] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*. IEEE, 999–1010. https://dl.acm.org/doi/abs/10.1145/3377811.3380386

[67] Qinying Wang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Yuhong Kan, Zhaowei Lin, Changting Lin, Shuiguang Deng, Alex X. Liu, and Raheem Beyah. 2021. MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols. In *proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*. USENIX Association, 4205–4222. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-qinying

[68] Zi Wang, Ben Liblit, and Thomas W. Reps. 2020. TOFU: Target-Orienter FUzzer. *CoRR* abs/2004.14375 (2020). https://arxiv.org/abs/2004.14375

[69] Nan Yang, Kousar Aslam, Ramon Schiffelers, Leonard Lensink, Dennis Hendriks, Loek Cleophas, and Alexander Serebrenik. 2019. Improving model inference in industry by combining active and passive learning. In *proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER 2019)*. IEEE, 253–263. https://ieeexplore.ieee.org/document/8668007

[70] Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. 2018. Symbolic verification of regular properties. In *proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. IEEE, 871–881. https://dl.acm.org/doi/10.1145/3180155.3180227

[71] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular property guided dynamic symbolic execution. In *proceedings of the 37th IEEE International Conference on Software Engineering (ICSE 2015)*. IEEE, 643–653. https://dl.acm.org/doi/10.5555/2818754.2818833