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
ljl@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 checking, runtime verification and greybox 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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/authors.
ACM ISBN 978-1-4503-9221-1/22/05.
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 \( \phi \), we construct the Büchi automata \( A_{\neg \phi} \) accepting \( \neg \phi \), and use this automata to guide the fuzz campaign. Thus, given a random input exercising an execution trace \( \pi \), we can check the “progress” of \( \pi \) in reaching the accepting states of \( A_{\neg \phi} \), and derive from \( A_{\neg \phi} \) the events that are needed to make further progress in the automata. Furthermore, in general, traces accepted by \( A_{\neg \phi} \) 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 \( A_{\neg \phi} \) 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 violations 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 fuzzing 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 data-set 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 \( \phi \) as inputs. The atomic propositions in \( \phi \) 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 \( \phi \) – 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 \( \phi \). 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 \( \phi \) as

\[
\neg F(a \land F(o \land G\neg n)) \tag{1}
\]

The negation of \( \phi \) is thus

\[
F(a \land F(o \land G\neg n))
\]

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 \( \phi \) 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/
example is the proposition \( \text{quota\_activated} = \text{true} \), which corresponds to the program location where quota checking is enabled in Pure-FTPd. At another statement, \( \text{user\_dir\_size} > \text{user\_quota} \), 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 \( \text{msg\_quota\_exceeded} = \text{true} \), we identify function invocations of addr_reply(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 \((l, p, c_p)\) 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': {
6067 user_quota_size *= (1024 ULL * 1024 ULL);
6068 if( liveness ) record_state();
6069 generate_event("a");
6070 if(liveness) record_state();
6071 }
```

### 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("a");
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(!)
4446 generate_event("n");
4447 if(liveness) record_state();
4448 }
```

### Listing 4: Entry of a loop statement:<ftpd.c, 4067>.
```
4066 for (;;) {
```

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Atomic Prop</th>
<th>Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>quota_activated = true</td>
<td>(a)</td>
<td>(ftpd.c, 6072)</td>
</tr>
<tr>
<td>user_dir_size &gt; user_quota</td>
<td>(o)</td>
<td>(safe_rw.c, 12)</td>
</tr>
<tr>
<td>msg_quota_exceeded = true</td>
<td>(n)</td>
<td>(ftpd.c, 4444, 43481)</td>
</tr>
<tr>
<td>loop_entry = true</td>
<td>(l)</td>
<td>(ftpd.c, 4071)</td>
</tr>
</tbody>
</table>

### Figure 1: Büchi automata accepting traces satisfying \(\neg\phi\).
```
18 + }
4867 + if(1){
4868 + generate_event("l");
4869 + if(liveness) record_state();
4870 + }
```

### 2.2 Program Transformation

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

**Event Generator.** To detect changes in \(\phi\)'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 \((l, p, c_p)\) \(\in L\), the generator injects a statement \(\text{if}(c_p) \text{generate\_event}("p")\); 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 \((\text{ftpd.c}, 6072)\) corresponds to the proposition variable \(a\) (\(\text{quota\_activated} = \text{true}\)) and the enabling condition is \(true\). The generator then inserts a statement \(\text{if}(1) \text{generate\_event}("a")\); at line 6072 in ftpd.c (see Listing 1). Consequently, whenever \((\text{ftpd.c}, 6072)\) 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 \(\phi\) at runtime. Specifically, the monitor produces a trace \(\tau\) by collecting events that are generated during execution (by the instrumented code). It then converts the negation of \(\phi\) to a Büchi automata \(A_{\neg\phi}\), and checks whether \(A_{\neg\phi}\) accepts \(\tau\). If the trace is accepted, the monitor reports a failure, i.e., \(\phi\) does not hold in \(P\). In our Pure-FTPd example, the negation of \(\phi\) is \(F(a \land F(o \land G\neg n))\), and the converted Büchi automata \(A_{\neg\phi}\) is illustrated in Figure 1.

**Checking Safety Properties.** A Büchi automata accepts a trace \(\tau\) if and only if \(\tau\) visits an accepting state of the automata “infinitely often” (e.g., state 2 in Figure 1). For the negation of a safety property \(\neg\delta\), the Büchi automata \(A_{\neg\delta}\) 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 \(A_{\neg\delta}\).
Checking Liveness Properties. The Büchi automata of the negation of $\phi$ accepts a trace $\tau$ if and only if $\tau$ visits an accepting state of $\mathcal{A}_{\neg\phi}$ “infinitely often” (e.g., state 2 in Figure 1). For instance, an infinite trace $a, o, (o)^\omega$ in which $o \neq \pi$ will be accepted by $\mathcal{A}_{\neg\phi}$. Formally, such a trace has the form $\tau = t_1(t_2)^\omega$ if $[t_2] \neq 0$, where $t_1$ 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 $t_2$ runs from the accepting state $s$ back to itself. Witnessing a trace $\tau = t_1(t_2)^\omega$ in which $t_2$ 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 $m$ times, e.g., $\tau = t_1, t_2t_3 \cdots$. This approach is insufficient because witnessing $t_2$ for $m$ times does not guarantee $t_2$ occurs infinitely often, for instance for $t = (1\pi; 1\pi\pi+2; 1\pi\pi+\cdots)\tau_2 \cdots$ may generate $t_2$ for $m$ times but stops generating $t_2$ after $t = m+1$.

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 $t_2$, that means, trace $t_2$ 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 $t_1(t_2)^\omega$ shaped trace.

Consider following two sequences witnessed in the execution

$$
\tau_e = e_0\epsilon_1 \cdots e_i\epsilon_{i+1} \cdots e_j\epsilon_j \cdots e_i\epsilon_i h
$$

$$
\tau_s = s_0s_1 \cdots s_i s_i h \cdots s_i h + 1 \cdots s_i h + 2
$$

where $\tau_e$ is a sequence of events associated with atomic propositions that occur in the execution and $\tau_s$ 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 $s_i \cdots s_{i+h}$ 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, $e_i\epsilon_{i+1} \cdots e_{i+h}$ can be generated infinitely many times by repeatedly executing input $I_{s_i \cdots s_{i+h+1}}$ on state $s_i$. Trace $\tau_e = e_0\epsilon_1 \cdots e_{i-1}(e_i \cdots e_h + \cdot \cdot \cdot)$ can be generated by running input $I_{s_i \cdots s_{i+h}}$ on state $s_i$, where $I_{s_i \cdots s_{i+h+1}}$ 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 $t_1(t_2)^\omega$ 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 $t_1(t_2)^\omega$ 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 transition 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.

3 BÜCHI AUTOMATA GUIDED FUZZING

Given an LTL property $\phi$ 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 $l_1$ that leads to $l_1$. Next, it takes $l_2$ as the target, and focuses on exploring the space of inputs starting with prefix $l_1$, i.e., keeping generating inputs starting with $l_1$. 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 $A_{\neg \phi}$ and the input prefix that leads to those transitions. Consider input $i_0$ and its trace $s_0$ goes from initial state $s_0$ to state $s_m$ on automata $A_{\neg \phi}$.

The achieved progress is represented as a tuple $(x^i, x^f)$, where $x^i$ is the shortest prefix of $i_0$ whose execution transition goes from $s_0$ to $s_m$ and $x^f$ is the state transition sequence $s_0 \cdots s_m$ visited. Such progress tuples are stored into a set $X$ and are used to guide fuzzing.

For input generation, the approach takes a tuple from $X$ and uses it to generate inputs that makes further progress. Consider a tuple $(x^i, x^f)$: $x^i$ records state transitions on automata $A_{\neg \phi}$ which input prefix $x^f$ has led to. Thus, we can query $A_{\neg \phi}$ with $x^i$ to find a transition that makes further progress, i.e., a state transition that gets closer to an accepting state of $A_{\neg \phi}$. In the example, assuming $x^i$ 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 $x^i$, then we can further query $A_{\neg \phi}$ 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 $\phi$ triggers transition from state 0 to state 1 and its corresponding program location is $(ftpd.c, 6072)$, 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 $x^f$ has gone through such that the generated trace can go through state transitions $x^i$; and, (2) subsequently the execution reaches one of program locations that are identified above to ensure the generated trace takes a step further in $A_{\neg \phi}$.

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 $A_{\neg \phi}$. 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 $X$ to generate inputs that make further "progress" towards an accepting state in the Büchi automata $A_{\neg \phi}$. 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 $A_{\neg \phi}$. Specifically, if the trace of the prefix of an input reaches a state that is closer to an accepting state on $A_{\neg \phi}$, 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 $A_{\neg \phi}$. Given an input prefix, the algorithm finds a state transition $t$ that helps us get closer to an accepting state in $A_{\neg \phi}$, 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 $A_{\neg \phi}$. 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 $target\_time$.

For prefix selection (line 5), the algorithm defines a fitness function to compute a fitness value for each prefix tuple. Given a tuple $(x^i_j, x^f_j)$, its fitness value is

$$f_i = \frac{l_x}{l_x + l_a} + \frac{1}{l_t},$$

where $l_x$ is the length of $x^i_j$ and $l_a$ is the length of the shortest path from the last state of $x^i_j$ to an accepting state on $A_{\neg \phi}$ and $l_t$ is the length of input prefix $x^f_j$. As shown in the formula, a prefix tuple has a higher fitness value if the last state of $x^i_j$ is closer to an accepting state on $A_{\neg \phi}$ 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 $A_{\neg \phi}$. Tuples with higher fitness values are prioritized for selection.

For atomic proposition selection (line 6), we adopt a random selection strategy. Consider tuple $(x^i_k, x^f_k)$ and the last state of $x^i_k$ is $s_k$, the algorithm identifies atomic propositions that make a progressive transition from $s_k$ on $A_{\neg \phi}$ as follows: if state $s_k$ is not an accepting state of $A_{\neg \phi}$, any atomic proposition that triggers a transition from $s_k$ towards an accepting state is selected. If state $s_k$ is an accepting state, any atomic proposition that triggers a transition from $s_k$ 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 negatives. 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 positives. 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 $\mathcal{A}_\phi$, where $\phi$ is the liveness property being checked. Note that a transition label in $\mathcal{A}_\phi$ is a subset of atomic propositions [62, 63]. The full set of atomic propositions is constructed by taking the atomic propositions appearing in $\phi$ 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 $l$ in our running example, we only store states for the program locations corresponding atomic proposition $l$ in Table 1.

Listing 5: Quota checking<-ftpdc.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 (ftpdc.c. 4067) (shown in Listing 4) that has been observed before and at the same time the witnessed trace is accepted by $\mathcal{A}_\phi$. In this case, LTL-Fuzzer reports a violation of the LTL property $\phi$ 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.
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.

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 direct 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.
- LTL-AFL. It is an implementation which enables AFLGo to detect an LTL property violation. Specifically, LTL-AFL 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 LTL-AFL, we evaluate how effective is our automata-guided fuzzing strategy in finding LTL property violations. Note that LTL-AFL 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].

Table 2: Detailed information about our subject programs.

<table>
<thead>
<tr>
<th>Project</th>
<th>Protocol</th>
<th>#SLOC</th>
<th>InPreviousWork</th>
<th>GithubStars</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProFTPD</td>
<td>FTP</td>
<td>210.8k</td>
<td>[52]</td>
<td>339</td>
</tr>
<tr>
<td>Pure-FTPd</td>
<td>FTP</td>
<td>52.9k</td>
<td>[52]</td>
<td>435</td>
</tr>
<tr>
<td>Live555</td>
<td>RTSP</td>
<td>52.5k</td>
<td>[52]</td>
<td>526</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>TLS</td>
<td>286.7k</td>
<td>[40]</td>
<td>16.3K</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>SSH</td>
<td>98.3k</td>
<td>[32]</td>
<td>1.5K</td>
</tr>
<tr>
<td>TinyDTLS</td>
<td>DTLS</td>
<td>63.2k</td>
<td>[31]</td>
<td>43</td>
</tr>
<tr>
<td>Contiki-Telnet</td>
<td>TELNET</td>
<td>333.4k</td>
<td>[40]</td>
<td>3.4K</td>
</tr>
</tbody>
</table>

4https://github.com/ltlfuzzer/LTL-Fuzzer/tree/main/ltl-property
5https://arxiv.org/abs/2109.02312
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], Sea-horn [36], SMACK [55], UAutomizer [38], DIVINE [14] cannot support LTL property verification. Schmell’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 utility. 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 AFL4LTL, 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.

<table>
<thead>
<tr>
<th>Prop</th>
<th>CVE-ID</th>
<th>Type of Vulnerability</th>
<th>Program Version</th>
<th>LTL-Fuzzer Time(h)</th>
<th>AFL$_{4LTL}$ Time(h)</th>
<th>AFLGo Time(h)</th>
<th>L$_{NuSMV}$ Time(h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>CVE-2019-18217</td>
<td>Infinite Loop</td>
<td>ProFTPD 1.3.6</td>
<td>4.62</td>
<td>T/O</td>
<td>1.00</td>
<td>T/O</td>
</tr>
<tr>
<td>$P_2$</td>
<td>CVE-2019-12815</td>
<td>Illegal File Copy</td>
<td>ProFTPD 1.3.5</td>
<td>0.95</td>
<td>2.01</td>
<td>0.84</td>
<td>T/O</td>
</tr>
<tr>
<td>$P_3$</td>
<td>CVE-2015-3306</td>
<td>Improper Access Control</td>
<td>ProFTPD 1.3.5</td>
<td>1.14</td>
<td>1.89</td>
<td>0.76</td>
<td>T/O</td>
</tr>
<tr>
<td>$P_4$</td>
<td>CVE-2010-3867</td>
<td>Illegal Path Traversal</td>
<td>ProFTPD 1.3.3</td>
<td>2.06</td>
<td>5.17</td>
<td>0.85</td>
<td>T/O</td>
</tr>
<tr>
<td>$L_1$</td>
<td>CVE-2019-6256</td>
<td>Improper Condition Handle</td>
<td>Live555 2018.10.17</td>
<td>5.29</td>
<td>11.13</td>
<td>1.00</td>
<td>11.47</td>
</tr>
<tr>
<td>$L_2$</td>
<td>CVE-2019-15232</td>
<td>Use after Free</td>
<td>Live555 2019.02.03</td>
<td>0.22</td>
<td>1.42</td>
<td>0.91</td>
<td>1.46</td>
</tr>
<tr>
<td>$L_3$</td>
<td>CVE-2019-7314</td>
<td>Use after Free</td>
<td>Live555 2018.08.26</td>
<td>1.27</td>
<td>4.18</td>
<td>0.98</td>
<td>T/O</td>
</tr>
<tr>
<td>$L_4$</td>
<td>CVE-2013-6934</td>
<td>Numeric Errors</td>
<td>Live555 2013.11.26</td>
<td>2.73</td>
<td>2.58</td>
<td>0.40</td>
<td>2.21</td>
</tr>
<tr>
<td>$L_5$</td>
<td>CVE-2013-6933</td>
<td>Improper Operation Limit</td>
<td>Live555 2011.12.23</td>
<td>1.80</td>
<td>1.99</td>
<td>0.63</td>
<td>1.45</td>
</tr>
<tr>
<td>$S_1$</td>
<td>CVE-2018-15473</td>
<td>User Enumeration</td>
<td>OpenSSH 7.7p1</td>
<td>0.18</td>
<td>0.17</td>
<td>0.44</td>
<td>T/O</td>
</tr>
<tr>
<td>$S_2$</td>
<td>CVE-2016-6210</td>
<td>User Information Exposure</td>
<td>OpenSSH 7.2p2</td>
<td>0.19</td>
<td>0.19</td>
<td>0.50</td>
<td>T/O</td>
</tr>
<tr>
<td>$S_3$</td>
<td>CVE-2016-6309</td>
<td>Use after Free</td>
<td>OpenSSL 1.1.0a</td>
<td>3.77</td>
<td>6.00</td>
<td>0.74</td>
<td>6.58</td>
</tr>
<tr>
<td>$S_4$</td>
<td>CVE-2016-6305</td>
<td>Infinite Loop</td>
<td>OpenSSL 1.1.0</td>
<td>1.45</td>
<td>T/O</td>
<td>1.00</td>
<td>T/O</td>
</tr>
<tr>
<td>$S_5$</td>
<td>CVE-2016-6306</td>
<td>Illegal Memory Access</td>
<td>OpenSSL 1.0.1f</td>
<td>1.11</td>
<td>7.31</td>
<td>1.00</td>
<td>T/O</td>
</tr>
</tbody>
</table>

For some tools, the LTL checker module is not available for usage / experimentation, as our email enquiry with CPAchecker team revealed.
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 $A_{12}$ [64] to evaluate whether one tool significantly outperforms another in terms of the time that is used to find a violation. The $A_{12}$ 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 $A_{12}$ values in bold.

LTL-Fuzzer found violations of all of the 14 LTL properties, followed by AFL$_{\text{LTL}}$ (12), AFLGo (5), and L+NuSMV (2). We note that AFL$_{\text{LTL}}$ 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$_{\text{LTL}}$ (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$_{\text{LTL}},$ AFLGo, and L+NuSMV, respectively. For CVE-2013-6934 and CVE-2013-6933, AFLGo performed slightly better than other techniques, while AFL$_{\text{LTL}}$ 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 $A_{12}$ statistic, LTL-Fuzzer performs significantly better than other techniques in most cases.

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 RFCs). For these properties, LTL-Fuzzer found violations of all of the 14 LTL properties in the experiment. AFL$_{\text{LTL}},$ AFLGo and L+NuSMV found 12, 5, 2 property violations, respectively. LTL-Fuzzer is 3.44x, 8.93x, 12.55x faster than AFL$_{\text{LTL}},$ AFLGo, and L+NuSMV.
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 $\Pr F_1$, $SL_2$, $TD_1$, and $PuF_3$. 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 $A_{12}$. 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], CPChecker [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 originates 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
We present LTL-Fuzzer (LTL) properties extracted from informal requirements such as RFCs to validate arbitrarily large and complex 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


Rujie Meng, Zhen Dong, Jinlin Li, Ivan Beschastnikh, and Abhik Roychoudhury