Wireless Protocol Validation Under Uncertainty

No Institute Given

Abstract. Validating wireless protocol implementations cannot always employ direct instrumentation of the device under test (DUT). The DUT may not implement the required instrumentation, or the instrumentation may alter the DUT's behavior when enabled. Wireless sniffers can monitor devices that direct instrumentation cannot, but they introduce new validation challenges. Losses caused by wireless propagation mean that even multiple sniffers cannot perfectly reconstruct the actual DUT packet trace. As a result, accurate validation using wireless sniffers requires distinguishing between specification deviations that represent implementation errors from those caused by sniffer uncertainty.

We present a new approach that enables sniffer-based validation of wireless protocol implementation. Beginning with the original protocol state machine, we automatically encode uncertainty introduced due to the sniffer by selectively adding non-deterministic transitions. Processing the sniffer packet trace using this validation state machine produces a set of mutated packet traces, each of which could have been actually observed by the DUT. We characterize the NP-completeness of the resulting problem and provide an exhaustive search algorithm for locating all mutated traces, as well as a more practical protocol-oblivious heuristics for locating likely mutated traces guided by the sniffer's packet loss characteristics. The resulting set of mutated traces can be used to determine whether the protocol implementation violates the specification. We have implemented our framework and show that it can accurately identify implementation errors.

1 Introduction

Custom wireless protocols are being designed and deployed to meet the specific performance and power needs of special-purpose wireless devices such as wearables [11], game controllers, and mobile computing devices. Validating that these devices correctly implement the protocol is crucial to achieve the design goals of the protocol and also prevent bugs in shipping products (e.g., Apple AWDL protocol bug in iOS 8 [6], Wi-Fi issues in Android Lollipop [9] and Microsoft Surface 3 [7]).

But validating the protocol implementation on such devices is challenging because collecting traces from the device under test (DUT) is often infeasible. The resource limitations of embedded or battery-powered devices may cause them to not provide trace collecting capabilities. Devices may include proprietary hardware or firmware that hides the protocol details. This may occur when multiple devices from different vendors are being tested for interoperability, or when development has been performed by a third-party that considers the implementation proprietary. And even if direct instrumentation is possible, the overhead it causes may alter the behavior of the DUT [19], threatening the validation results.

An attractive alternative is to use wireless sniffers to record traffic generated by the DUTs during testing. Sniffers do not require direct access to the DUT or alter its behavior. However, due to the fundamentally unpredictable nature of wireless communications, the packets captured by the sniffer will not exactly match those received by the DUT. The sniffer may miss packets that the DUT received, or receive packets that the DUT missed. This is true even when using multiple sniffers [5,17,3], sniffer with multiple antennas [21], or in isolated wireless environments. (Isolated environments are also inappropriate for testing in the common case when the DUT must cope with interference encountered in uncontrolled wireless environments.)

Because the sniffer trace does not perfectly match the actual trace, uncertainty arises during protocol implementation validation. For example, if the DUT fails to respond correctly to a packet in the sniffer trace, it may be because (a) the DUT's implementation is incorrect, (b) the DUT did not actually receive the packet or (c) the DUT's response was missed by the sniffer. Whenever the DUT's behavior does not match the specification, there are now two potential explanations: either the DUT's implementation is wrong, or the sniffer trace is incorrect. Accurate validation requires accurately distinguishing between these two causes.

We present a new technique enabling accurate validation of protocol implementation using wireless sniffers. Given a state machine representing the protocol being validated, we describe a systematic transformation that adds nondeterministic transitions to incorporate uncertainty introduced by the sniffer. This augmented validation state machine processes the sniffer trace into a set of mutated traces, each satisfying the original state machine with certain probability. If the set is empty, the implementation definitely violates the protocol. If the set contains only low-probability traces, then the implementation probably violates the protocol. Finding all mutated traces is NP-complete, but the approach can be made practical by applying protocol-oblivious heuristics that limit the search to likely mutated traces.

Our paper makes the following contributions:

- To the best of our knowledge we are the first to identify the uncertainty problem caused by sniffer in validating wireless protocol implementations.
- We formalize the problem using a nondeterministic state machine that systematically and completely encodes the inherent uncertainty of the sniffer trace.
- We characterize the NP-completeness of the validating problem, and present protocol-oblivious heuristics to prune the search space and make validation possible in practice.
- We implement the validation framework and evaluate it using NS-3 network simulator [20]. Our framework accurately identifies both introduced as well

as previously unknown violations in NS-3's implementations of the 802.11 protocol.

The rest of this paper is organized as follows. We motivate the uncertainty problem in Section 2. We then formally describe the problem in Section 3, including the completeness of the augmentation (§ 3.2), hardness analysis (§ 3.3) and search algorithms (§ 3.4). We continue by briefly describe our implementation in Section **??** and evaluate our framework through two case studies in Section 4. Finally, we present related works in Section 5 and concludes in Section 6.

2 Uncertainty of Sniffer Trace

There are two fundamental properties in wireless communication: non-zero packet loss probability and physical diversity. Suppose the DUT communicates with an endpoint using certain protocol, the non-zero packet loss property means the sniffer could miss packets from the DUT or the endpoint. Physical diversity property means each device's packet reception probability is independent with each other. Therefore, a sniffer could either miss packets seen by the DUT, or capture extra packets that are missed by the DUT. It is precisely such incompleteness of sniffer packet traces that brings uncertainty to the validation problem.

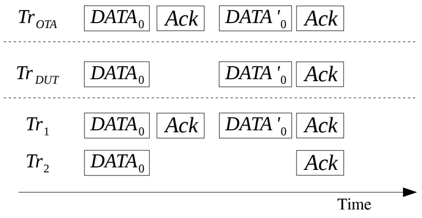


Fig. 1: **Uncertainty Caused by Sniffer Observations.** Tr_{OTA} is the chronological sequence of packets sent by the DUT and the endpoint. Tr_{DUT} is the trace of the DUT. Tr_1 and Tr_2 are two possible traces of the sniffer.

Consider the 802.11 data transmission protocol and a few example traces observed by sniffers shown in Fig. 1. The transmitter (DUT) sends a packet $DATA_0$. Even though the receiver (endpoint) receives $DATA_0$ and sends out an *Ack* packet, the DUT does not receive the *Ack* packet. Eventually the acknowledgment timer fires and the DUT retransmits $DATA_0$ (denoted as $DATA'_0$) and receives the acknowledgment. It is obvious that the DUT obeys the protocol with respect to its own observation Tr_{DUT}.

In first possible sniffer trace Tr_1 where the sniffer *overhears* the first *Ack* packet, a validation *uncertainty* arises when the sniffer sees the retransmission packet $DATA'_0$: was the previous *Ack* missed by the DUT or is there a bug in DUT which causes it to retransmit even after receiving the *Ack*? Similarly, consider another possible sniffer trace Tr_2 where both the $DATA'_0$ and *Ack* packets were missed by the sniffer. During this period of time, it appears the DUT neither receives acknowledgment for $DATA_0$ nor retransmits $DATA'_0$. Again, without any additional information it is impossible to disambiguate between the sniffer missing certain packets and a bug in DUT's retransmission logic.

Assume that the endpoint implementation is correct, **given the expected behavior of the DUT and the sniffer's observation with inherent un-**

certainty, can we validate that the DUT behaves as specified by the protocol? This is the question we set out to answer in this paper.

3 Verification Framework

We first present the formal framework for validating DUT implementation in the absence of uncertainty. We then propose a systematic augmentation scheme to encode the uncertainty caused by sniffer traces.

3.1 Packet, Trace and Monitor

The observations of the DUT constitute packets sent and received at certain timestamps. Aspects of the protocol specification can be monitored using state machines to validate the observations of the DUT. In this section, we formally define these concepts. More specifically, we use *timed automata* [1] to model the expected behaviors of the DUT with correct protocol implementation. A timed automata is a finite state machine with timing constraints on the transitions: each transition can optionally start one or more timers, which can later be used to assert certain events should be seen before or after the time out event. We use the term *timed automata* and *state machine* interchangeably hereafter.

The input alphabet of the state machine is the finite set of all valid packets defined by the protocol, denoted as $\mathbb{P}$. A packet is a binary string of finite number of bits, encoding interesting protocol attributes such as `src`, `dest`, `type`, `flags`, and physical layer information, such as `channel`, `modulation`, etc.

Next, we define the *observation* that corresponds to time-ordered sequence of packets.

Definition 1. *A packet trace is a finite sequence of* $(timestamp, packet)$ *tuple:*

$$[(t_1, p_1), (t_2, p_2), \ldots, (t_n, p_n)]$$

where $t_i \in \mathbb{Z}^+$ *is the discrete timestamp and* p_i *is the packet observed at time* t_i*. The timestamps are strictly monotonically increasing,* $t_i < t_{i+1}$ *for* $1 \leq i < n$*.*

In addition to timestamp monotonicity, we also require that adjacent packets do not overlap in time, $t_{i+1} - t_i > \texttt{airtime}(p_i)$ for $1 \leq i < n$, where `airtime()` calculates the time taken to transmit a packet based on its size and modulation scheme.

We now formally define the monitor state machine that models the expected behavior of the DUT and serves as the protocol specification.

Definition 2. *A protocol monitor state machine* S *is an 6-tuple* $\{\Sigma, \mathbb{S}, s_0, C, E, G\}$*, where:*

- $\Sigma = \mathbb{P}$ *is the finite input alphabet.*
- $\mathbb{S}$ *is a non-empty, finite set of states.* $s_0 \in \mathbb{S}$ *is the initial state.*

- *C is the set of clock variables. A clock variable can be reset along any state transitions. At any instant, reading a clock variable returns the time elapsed since last time it was reset.*
- *G is the set of clock constraints defined inductively by*

$$g := true \mid c \leq T \mid c \geq T \mid \neg g \mid g_1 \wedge g_2$$

  where $c \in C$ is a clock variable and T is a constant. Note that an transition can choose not to use clock guard by setting g to be true.
- *$E \subseteq \mathbb{S} \times \mathbb{S} \times \Sigma \times G \times \mathscr{P}(C)$ gives the set of transitions. $\langle s_i, s_j, p, g, C' \rangle \in E$ represents that if the monitor is in state s_i, given the input tuple (t, p) such that the clock variables satisfies the guard g, the monitor transits to state s_j and reset the clocks in C' to 0.*

A tuple (t_i, p_i) in the packet trace means the packet p_i is presented to the state machine at time t_i. The monitor rejects a trace Tr if there exists a prefix of Tr such that all states reachable after consuming the prefix have no valid transitions for the next (t, p) input. Note that the input to the state machine, time and packet, are all observable events in the communication medium. Validating the internal events of the DUT is beyond the scope of this paper.

Next, we give an example monitor state machine for the transmitter in 802.11 data transmission protocol. Without loss of generality, we assume the DUT only retransmits the packet at most once if the first transmission failed. The state machine can be easily adapted to accommodate multiple retransmissions.

There are only three types of packets to consider in this case: $DATA_i$ denotes a packet sent by the transmitter with sequence number i, $DATA'_i$ denotes the retransmission of $DATA_i$, and Ack is the acknowledgment packet sent by the receiver. The sequence number is within the range of $[0, N)$.

The monitor state machine is illustrated in Fig. 2, and the main components are:

- $\Sigma = \{DATA_i, DATA'_i, Ack \mid 0 \leq i < N\}$.
- Clock variables $C = \{c\}$. The only clock variable c is used for acknowledgment time out.
- Guard constraints $G = \{c \leq T_o, c > T_o, T_o < c \leq T_m\}$. T_o is the acknowledgment time out value, and $T_m > T_o$ is the maximum delay allowed before the retransmission packet gets sent. T_o can be arbitrary large but not infinity in order to check the liveness of the DUT.

To succinctly present the state machine, we use an internal variable i to keep track of the next sequence number. One can easily eliminate this variable and use only different states for different sequence numbers. Therefore, this state machine is consistent with Definition 2.

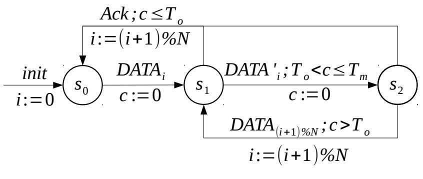


Fig. 2: **Monitor State Machine for 802.11 Packet Transmitter.**

The sequence number i is initialized to 0 when the state machine is initialized. At state s_0, the monitor expects to see a $DATA_i$ packet. It transits to s_1 when such event happens and resets acknowledgment timer c along the transition. At state s_1, two valid events can happen: either the DUT receives Ack packet within time T_o ($s_1 \rightarrow s_0$), or the DUT sends $DATA'_i$ after T_o but before T_m ($s_1 \rightarrow s_2$). Similarly at state s_2, either the DUT receives Ack within time T_o ($s_2 \rightarrow s_0$), or otherwise the DUT concludes that the previous transmission failed and continues to transmit the next packet $s_2 \rightarrow s_1$.

The monitor state machine defines a *timed language* L which consists all valid packet traces that can be observed by the DUT. We now give the definition of protocol *compliance* and *violation*.

Definition 3. *Suppose* $\mathbb{T}$ *is the set of all possible packet traces collected from DUT, and* S *is the state machine specified by the protocol. The DUT violates the protocol specification if there exists an packet trace* $Tr \in \mathbb{T}$ *such that* S *rejects* Tr*. Otherwise, the DUT is compliant with the specification.*

The focus of this paper is to determine whether a *given* Tr is evidence of a violation. We acknowledge that determining *compliance* is a more challenging problem, as it requires enumerating every possible trace in $\mathbb{T}$, which is probably infinite.

3.2 Augmented State Machine

To deal with the inherent uncertainty of sniffer traces, we propose to systematically augment the original checker state machine with non-deterministic transitions to account for the difference between the sniffer and DUT traces.

Before formally defining the augmented state machine, we first use an example to illustrate the basic idea. Fig. 3 shows the augmented state machine for 802.11 transmitter state machine shown in Fig. 2. For each existing transition (e.g., $s_0 \rightarrow s_1$), we add an *empty transition* with same clock guards and resetting clocks. This is to account for the possibility when such packet was observed by the DUT but missed by the sniffer. Additionally, for each transition triggered by a *receiving* packet (i.e., $p.dest = DUT$), such as $s_1 \rightarrow s_0$ and $s_2 \rightarrow s_0$, we add a *self transition* with the same trigger packet and clock guards, but empty set of resetting clocks. The presence of this transition allows the protocol to make progress on a trace, and not get stuck

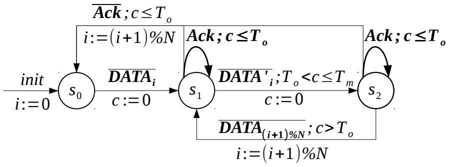


Fig. 3: **Augmented Checker State Machine.** Augmented transitions are highlighted in bold face. $\overline{Pkt}$ means either ϵ or Pkt.

There are two points worth noticing. First, self transitions are added only for receiving packets with respect to the DUT. If the sniffer observes a sending packet from the DUT, then the packet must also appear in the DUT's observation. In other words, there is no uncertainty for such packets when they are observed by the sniffer. Second, no augmented transition are added for the packets that are sent to DUT yet are missed by both the DUT and the sniffer, since such packets do not cause difference between the DUT and sniffer traces.

The augmented state machine in Fig. 3 will accept the sniffer packet traces Tr_1 and Tr_2 shown in Fig. 1. For instance, one accepting transition sequence on sniffer trace Tr_1 is $s_0 \to s_1 \to_s s_1 \to s_2 \to s_0$, and the sequence for Tr_2 is $s_0 \to s_1 \to_e s_2 \to s_0$, where $\to$ is the transition from the original state machine, $\to_e$ and $\to_s$ are the augmented empty and self transitions respectively.

We now formally define the augmented state machine.

Definition 4. *An augmented state machine S^+ for a checker state machine S is a 6-tuple $\{\boldsymbol{\Sigma^+}, \mathbb{S}, s_0, C, \boldsymbol{E^+}, G\}$, where $\mathbb{S}, s_0, C, G$ are the same with S. $\Sigma^+ = \{\epsilon\} \cup \Sigma$ is the augmented input alphabet with the empty symbol, and $E^+ \supset E$ is the set of transitions, which includes:*

- *E: existing transitions (**Type-0**) in S.*
- *E_1^+: empty transitions (**Type-1**) for each transition in E.*
- *E_2^+: self transitions (**Type-2**) for each transitions triggered by receiving packets.*

With augmented state machine S^+, we can use Type-1 transitions to non-deterministically infer packets missed by the sniffer, and use Type-2 transitions to consume extra packets captured by the sniffer but missed by the DUT.

Suppose P is an accepting transition sequence of sniffer trace Tr on the augmented state machine S^+. If we add missed packets indicated by Type-1 transitions to Tr, and remove packets indicated by Type-2 transitions from Tr, we obtain a *mutation* trace Tr', which represents one possibility of the ground truth—the DUT packet trace.

Note that Tr' may contain arbitrary number of packets that are not in Tr, but can only remove receiving packets from Tr, since Type-2 transitions are only for receiving packets. This relationship is formally expressed in the following definition.

Definition 5. *A packet trace Tr' is a mutation of sniffer trace Tr with respect to a DUT if for all $(t, p) \in Tr \setminus Tr'$, $p.dest = DUT$.*

A mutation trace Tr' represents a *guess* of the corresponding DUT packet trace given given sniffer trace Tr. In fact, the DUT packet trace must be one of the mutation traces of the Tr.

Lemma 1. *Let Tr_{DUT} and Tr be the DUT and sniffer packet trace captured during the same protocol operation session, and $\mathcal{M}(Tr)$ be the set of mutation traces of Tr with respect to DUT, then $Tr_{DUT} \in \mathcal{M}(Tr)$.*

Proof. Let $\Delta = Tr \backslash Tr_{DUT}$ be the set of packets that are in Tr but not in Tr_{DUT}. Recall that it is not possible for the sniffer to observe sending packets from the DUT that the DUT did not send. Therefore, all packets in Δ are receiving packets with respect to DUT. That is, for all $(t, p) \in \Delta$, $p.dest = DUT$. By Definition 5, $Tr_{DUT} \in \mathcal{M}(Tr)$.

Lemma 1 shows that $\mathcal{M}(Tr)$ is a *complete* set of guesses of the DUT packet trace. We now claim the satisfiability of mutation trace Tr' on the original state machine S based on the augmented state machine S^+ and the sniffer trace Tr.

Theorem 1. *There exists a mutation trace $Tr' \in \mathcal{M}(Tr)$ that satisfies S if and only if Tr satisfies S^+.*

Theorem 1 shows that in order to determine if the DUT *could have* behaved as specified (Tr_{DUT} satisfies S), we only need to determine if the sniffer trace Tr satisfies the augmented state machine S^+. The inherent uncertainty of the sniffer traces are explicitly represented by the augmented transitions, and can be systematically explored using the well established state machine theory.

One immediate observation can be drawn from Theorem 1 by contradiction.

Corollary 1. *If S^+ rejects Tr, then S rejects Tr_{DUT}.*

In the context of validation where we raise a violation alarm when S^+ rejects Tr, Corollary 1 guarantees that no false alarms will be raised. However, when S^+ accepts Tr, S could still reject Tr_{DUT}. In other words, the conclusion of the validation can either be *definitely wrong* or *possibly correct*, but not *definitely correct*. This is the fundamental limitation caused by the uncertainty of sniffer traces.

3.3 Problem Hardness

In this section, we show that the problem of determining whether S^+ accepts sniffer trace Tr is NP-complete. In fact, the problem is still NP-complete even with only one type of augmented transitions.

Recall that Type-1 transitions are added because the sniffer may miss packets. Suppose there is a special sniffer that is able to capture every packet transmitted in the communication medium. In this case, there is no need to add Type-1 transitions. However, Type-2 transitions are still needed since the sniffer may overhear packets sent to the DUT. Similarly, suppose another special sniffer that would not overhear any packets sent to the DUT. That is, when this sniffer observes receiving packets, the DUT must also have received the packet. Therefore, Type-2 transitions are no longer needed.

We refer the augmented state machine that only have Type-0 and Type-1 transitions as S_1^+, and the augmented state machine only have Type-0 and Type-2 transitions as S_2^+. Now we show that each subproblem of determining trace satisfiability is NP-complete.

Problem 1. VALIDATION-1
Given that $Tr \setminus Tr_{DUT} = \emptyset$.
instance Checker state machine S and sniffer trace Tr.
question Does S_1^+ accept Tr?

Problem 2. VALIDATION-2
Given that $Tr_{DUT} \subset Tr$.
instance Checker state machine S and sniffer trace Tr.
question Does S_2^+ accept Tr?

Lemma 2. *Both VALIDATION-1 and VALIDATION-2 are NP-complete.*

We now turn to the general validation problem with normal sniffers that could either miss or overhear packets.

Problem 3. VALIDATION
instance Checker state machine S and sniffer trace Tr.
question Does S^+ accept Tr?

Theorem 2. *VALIDATION is NP-complete.*

Proof. Based on the same reasoning that VALIDATION-1 and VALIDATION-2 are in NP, VALIDATION is also in NP. Furthermore, it is easy to see that both VALIDATION-1 and VALIDATION-2 are special instances of VALIDATION. Therefore, VALIDATION is NP-complete.

3.4 Searching Strategies

In this section, we present an *exhaustive* search algorithm of the accepting transition sequence of S^+ on sniffer trace Tr. It guarantees to yield a solution if exists, thus is exhaustive. Due to the NP-completeness of the problem, we also propose several heuristics in next section to limit the search to only mutations with high likelihood, which makes the algorithm meaningful in practice.

The main routines of the algorithm is shown in Algorithm 1. In the top level `SEARCH` routine, we first obtain the augmented state machine S^+, then we call the recursive `EXTEND` function with an empty prefix, the sniffer trace, and the state machine's initial state.

In the `EXTEND` function, we try to consume the first packet in the remaining trace using either Type-0, Type-1 or Type-2 transition. Note that we always try to use Type-0 transitions before other two augmented transitions (line 7). This ensures the first found mutation trace will have the most number of Type-0 transitions among all possible mutation traces. Intuitively, this means the search algorithm tries utilize the sniffer's observation as much as possible before being forced to make assumptions.

Each of the extend function either returns the mutation trace Tr', or *nil* if the search fails. In both `EXTEND-0` and `EXTEND-2` function, if there is a valid transition, we try to consume the next packet either by appending it to the prefix (line 16) or dropping it (line 32). While in `EXTEND-1`, we guess a missing packet without consuming the next real packet (line 24). Note that since only Type-0 and Type-2 consume packets, the recursion terminates if there is a valid Type-0 or Type-2 transition for the last packet (line 15 and line 31).

It is easy to see that Algorithm 1 terminates on any sniffer traces: each node in the transition tree only has finite number of possible next steps, and the

Algorithm 1 Exhaustive search algorithm of S^+ on Tr.

```
 1: function SEARCH(S, Tr)
 2:     $S^+$ := AUGMENT(S)
 3:     return EXTEND([], Tr, $S^+.s_0$)
 4: function EXTEND(prefix, p::suffix, $s$)
 5:     if not LIKELY(prefix) then  ▷ This check should be ignored in exhaustive alg.
 6:         return $nil$
 7:     for i ∈ [0, 1, 2] do
 8:         mutation := EXTEND-i(prefix, p::suffix, $s$)
 9:         if mutation $\neq$ $nil$ then
10:             return mutation
11:     return $nil$
12: function EXTEND-0(prefix, p::suffix, $s$)
13:     for $\langle s, s', p \rangle \in E$ do
14:         if suffix = $nil$ then
15:             return prefix@p
16:         mutation := EXTEND(prefix@p, suffix, $s'$)
17:         if mutation $\neq$ $nil$ then
18:             return mutation
19:     return $nil$
20: function EXTEND-1(prefix, p::suffix, $s$)
21:     for all $\langle s, s', q \rangle \in E_1^+$ do
22:         if q.time > p.time then
23:             continue
24:         mutation := EXTEND(prefix@q, p::suffix, $s'$)
25:         if mutation $\neq$ $nil$ then
26:             return mutation
27:     return $nil$
28: function EXTEND-2(prefix, p::suffix, $s$)
29:     for all $\langle s, s, p \rangle \in E_2^+$ do
30:         if suffix = $nil$ then
31:             return prefix
32:         mutation := EXTEND(prefix, suffix, $s$)
33:         if mutation $\neq$ $nil$ then
34:             return mutation
35:     return $nil$
```

depth of Type-1 transitions are limited by the time available before the next packet (line 22). Note that Type-1 transitions do not consume packets, but take certain amount of time to perform. The time taken depends on the duration of the inferred packets or the time constraints of the transition's clock guard.

3.5 Mutation Constraints

Fundamentally, augmented transitions are introduced to compensate the imperfection of sniffers. Whenever the original state machine S gets stuck on sniffer trace Tr, Algorithm 1 tries to continue with augmented transitions. In other words, in the face of uncertainty between a possible protocol violation and sniffer imperfection, augmented transitions provide the ability to blame the later. In fact, the exhaustive nature of Algorithm 1 means that it always tries to blame sniffer imperfection whenever possible, making it too conservative to report true violations.

Therefore, extra constraints on the mutation trace need to be enforced to restrict the search only to mutation traces with high likelihood. The modified `EXTEND` function checks certain likelihood constraints on the prefix of the mutation trace before continue (line 5), and stops the current search branch immediately if the prefix seems *unlikely*. Because of the recursive nature of the algorithm, other branches which may have a higher likelihood can then be explored.

The exact forms of the constraints may depend on many factors, such as the nature of the protocol, properties of the sniffer, or domain knowledge. Note that the strictness of the likelihood constraint represents a trade-off between precision and recall of violation detection: the more strict the constraints are, the more false positive violations will potentially be reported, hence the lower the precision yet higher recall. In the most extreme case when the constraint does not allow any augmented transitions at all, each missed or extra packets in the sniffer trace can result in a violation report. On the contrary, the more tractable the constraints are, the more tolerant the search is to sniffer imperfection, hence the more likely that it will report true violations, thus higher precision but lower recall.

Next, we propose two *protocol oblivious* heuristics based on the sniffer loss probabilities and general protocol operations. The first heuristic is $NumMissing(d, l, k)$, which states that the number of missing packets from device d in any sub-mutation traces of length l shall not exceed k ($k \leq l$). The sliding window of l serves two purposes. First, l should be large enough for the calculated packet loss ratio to be statistically meaningful. Second, it ensures that the packet losses are evenly distributed among the entire packet trace.

The intuition behind the $NumMissing$ heuristic is that sniffers can often be configured such that the packet loss probabilities are very low. For instance, multiple sniffers can be deployed around the DUT and the endpoint separately and the traces can be fused to provide a unified and potentially more complete view. Or the raw radio signals can be collected using SDRs and the packets can be reconstructed offline to improve capturing quality. In such cases, the likelihood that the sniffer misses too many packets is quite low. In other words, if a mutation trace contains too many Type-1 augmented transitions, then the likelihood that it is the observation of the DUT is very low, thus is not worth exploring during the search.

In addition, we note that for a fixed l, a single threshold k may not work for traces collected by sniffers with different packet loss probabilities. Intuitively, sniffers with high loss probabilities require a larger threshold, while sniffers with low loss probabilities may do with small thresholds. Therefore, instead of using a single constant k, we perform a *stratified* search with an increasing sequence of thresholds $\{k_1, k_2, \ldots, k_n\}$ ($k_i < k_{i+1}$ for $1 \leq i < n$). At round i, the search is restricted by constraint $NumMissing(d, l, k_i)$. If a mutation trace is found, the search is completed. Otherwise, we increase the round number and repeat the search process. In this way, the constraint is automatically adaptive to the underlying sniffer packet loss probabilities. And a violation is declared when no mutation trace can be found with $NumMissing(d, l, k_n)$.

The second heuristic is $GoBack(k)$, which states that the search should only back track at most k steps when gets stuck. The motivation is that many protocols operate in a sequence of independent transactions, and the uncertainty of previous transactions often do not affect the next transaction. For instance, in 802.11 packet transmission protocol, each packet exchange, include the original, retransmission and acknowledgment packets, constitute a transaction. And the retransmission status of previous packets has no effect on the packets with next sequence number, hence need not be explored when resolving the uncertainty of the packets with new sequence numbers.

4 Case Studies

We present three case studies on applying our validation framework on protocols implemented in the NS-3 network simulator (802.11 data transmission and ARF rate control). The goal is to demonstrate how our framework can avoid raising false alarms on incomplete sniffer traces and report true violations.

4.1 802.11 Data Transmission

We have three main goals in this section. First, we show that our verification framework can improve verification precision by inferring the missing or extra packets using the argumented transition framework. Second, we study how the packet loss ratios affect the quality of the verification. Finally, we demonstrate the ability of our framework to detect true violations by manually introducing bugs in the NS-3 implementation and by showing the precision and recall of violation detection.

Experimental Setup We set up one Wi-Fi client device and one Access Point (AP), which act as the DUT and endpoint respectively. Another Wi-Fi device is configured in monitor mode and acts as the sniffer. The DUT and endpoint are configured to use the IEEE 802.11g standard with both RTS/CTS and fragmentation disabled. A Constant Bit Rate (CBR) UDP traffic (54 Mbps) is generated from the DUT to the endpoint. The UDP packet size is 1436 bytes, which results in a 1500 bytes Wi-Fi packet.

In order to control the packet loss ratios between each pair of devices, we developed a new propagation loss model for NS-3 called `MatrixRandomPropagationLossModel`. Instead of a constant propagation loss as in existing `MatrixPropagationLossModel`, the signal propagation loss between a pair of nodes is determined by a binary random variable of two values: 0 dB (no loss) and 1000 dB (complete loss). Therefore, any packet loss probability can be achieved by adjusting the random variable distribution. The model supports both symmetric and asymmetric propagation losses. We use symmetric propagation loss in all our experiments. Finally, pcap capture is enabled in both the DUT and the sniffer devices.

Verifying Unmodified Implementation In NS-3's implementation of 802.11g protocol, the acknowledgment time out T_o is 334 μs and maximum retransmission count is 7. We use the same parameters in our checker state machine. Additionally, we set the maximum retransmission delay T_m to be 15 ms to tolerate the back off delay in 802.11 DCF and also any possible processing delays.

Let Pr_{ds}, Pr_{es} and Pr_{ed} be the packet loss probability between the DUT and sniffer, endpoint and sniffer, DUT and endpoint respectively. We vary each probability from 0 to 0.5 (both inclusive) with 0.05 step. For each loss ratio combination, we ran the experiment 5 times, and each run lasted 30 seconds. In total, 6655 ($11^3 \times 5$) pair of DUT and sniffer packet traces were collected.

To establish the ground truth of violations, we first verify the DUT packet traces using the *original* state machine S. This can be achieved by disabling augmented transitions in our framework. As expected, no violation is detected in any DUT packet traces.

We then verify the sniffer traces using the augmented state machine S^+. For the $NumMissing$ heuristic, we use $l = 100$ and $k = Pr \times r \times l$ for $1.0 \leq r \leq 2.0$ with 0.2 step, and Pr is Pr_{ds} for the DUT and Pr_{es} for the endpoint. For the $GoBack(k)$ heuristic, we set $k = 7$, which is the maximum number of transmissions of a single packet.

We first report that for all sniffer traces collected with different packet loss probabilities, no violation is reported, which demonstrates our framework's ability to tolerate sniffer imperfection instead of raising false alarms.

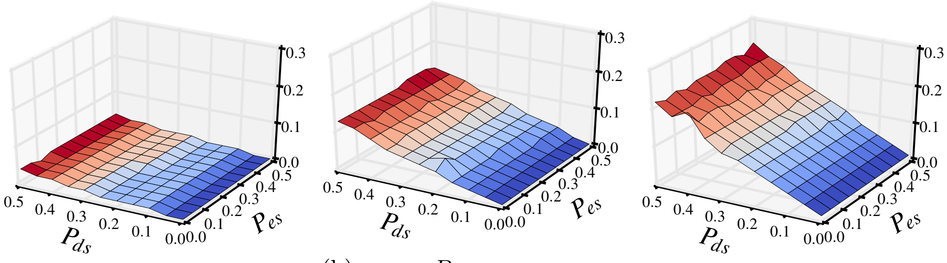
Next, we present detailed analysis of the augmented transitions on the sniffer traces. The goal is to study for a given link packet loss probability (Pr_{ed}), how the sniffer's packet loss properties (Pr_{ds} and Pr_{es}) affect the difference between mutation and the DUT trace. For all following analysis, we divide the traces into three groups according to Pr_{ed}: low ($0 \leq Pr_{ed} \leq 0.15$), medium ($0.20 \leq Pr_{ed} \leq 0.35$) and high ($0.40 \leq Pr_{ed} \leq 0.50$).

The different between two packet traces can be quantified by the Jaccard distance metric.

$$Jaccard(Tr_1, Tr_2) = \frac{Tr_1 \ominus Tr_2}{Tr_1 \cup Tr_2} \tag{1}$$

where $\ominus$ is the symmetric difference operator. The distance is 0 if the two traces are identical, and is 1 when the two traces are completely different. The smaller the distance is, the more similar the two traces are.

A naive way to calculate the distance is to use the hash the (t, p) pair for set intersection and union operation. However, it does not work for mutation trace which contains fabricated packets with no actual payload. Therefore, we



(a) $Pr_{ed} \in [0.05, 0.1, 0.15]$
(b) $Pr_{ed} \in [0.2, 0.25, 0.3, 0.35]$
(c) $Pr_{ed} \in [0.4, 0.45, 0.5]$

Fig. 4: **Jaccard Distance Between Mutation and DUT Traces.** For each data point, the mean of the 5 runs is used.

use a protocol specific canonical representation of packets when calculating the distance. In particular, the string `r_DATA_i_t` represents the t^{th} transmission of a data packet with sequence number i. r represents the round of sequence numbers as it wraps after 4096. And similarly `r_Ack_i_t` is the corresponding acknowledgment packet.

Figure 4 shows the Jaccard Distance between mutation and its corresponding DUT trace. We make the following observations. First, for a given system loss probability, the more packet the sniffer picks up, the more similar the mutation trace is to the DUT trace. Interestingly, Pr_{ds} is the dominant factor between the two. This is because retransmission packets of the same sequence number are identical, hence it may require to infer less retransmission packets than that were actually transmitted to unstuck the state machine. We note, however, this trend is protocol specific and may be not be generally applicate to other protocols. Second, as the system loss probability increases, the Jaccard distance increases more rapidly as Pr_{ds} increases. This is because the ratio of retransmission packet increases along with Pr_{ds}.

Introducing Bugs We have demonstrated that our framework can tolerate sniffer imperfection and avoid raising false alarms. The next question is, can it detect true violations? To answer this question, we manually introduce several bugs in NS-3 implementation that concerns various aspects of 802.11 data transmission protocol. More specifically, the bugs are:

- **Sequence Number** The DUT does not assign sequence number correctly. For example, it may increase sequence by 2 instead of 1, or it does not increase sequence number after certain packet, etc.
- **Semantic** The DUT may retransmit even after receiving *Ack*, or does not retransmit when not receiving *Ack*.

We instrument the NS-3 implementation to embed instances of bugs in each category. At each experiment run, we randomly decide whether and which bug to introduce for each category. We fix $Pr_{ds} = Pr_{es} = 0.1$ and vary Pr_{ed} from

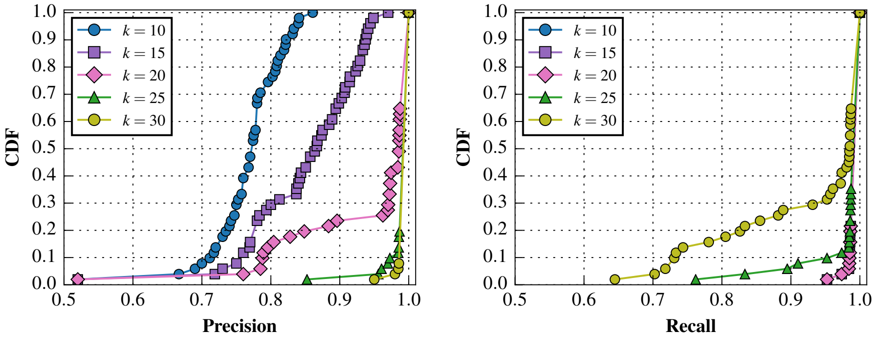


Fig. 5: **Precision and Recall of Violation Detection.**

0.0 to 0.5 with 0.01 step. For each Pr_{ed} value, we ran the experiment 100 times, of which roughly 75 experiments contained bugs. In total, 5100 pairs of DUT and sniffer traces were collected.

We use the DUT packet traces as ground truth of whether or not each experiment run contains bugs. For each Pr_{ed} value, we calculate the precision and recall of violation detection using the sniffer traces.

$$\text{Precision} = \frac{\{\text{Reported Bugs}\} \cap \{\text{True Bugs}\}}{\{\text{Reported Bugs}\}} \tag{2}$$

$$\text{Recall} = \frac{\{\text{Reported Bugs}\} \cap \{\text{True Bugs}\}}{\{\text{True Bugs}\}} \tag{3}$$

The precision metric quantifies how *useful* the validation results are , while the recall metric measures how *complete* the validation results are.

Figure 5 shows the validating precision and recall for various k values. As expected, the more tolerant the search to sniffer losses (larger k), the more precise the violation detection. In particular, when $k = 30$, the precisions are 100% for all Pr_{ed} values. Second, the recall is less sensitive to the choice of k. Except for the extreme case when $k = 30$, all other thresholds can report almost all the violations.

4.2 ARF Rate Control Algorithm

We report a bug found in NS-3 ARF [13] implementation which causes the sender stuck at a lower rate even after enough number of consecutive successes. This bug also exists in the NS-3 implementation of Adaptive ARF (AARF) algorithm [14], which is similar to ARF but use Binary Exponential Backoff to adjust the thresholds dynamically. The pseudo code implementation of AARF in [15] also suffers the same problem.

5 Related Works

HMM-Based Approach. Event sampling [4,12,2,8] reduces the overhead of performing runtime verification, yet creates gaps in the observation. Stroller *et al* [22] studied how event sampling affects the results of runtime verification, and use HMM-based state estimation techniques to calculate the confidence that the temporal property is satisfied in the presence of gaps in observation. When considering the packets as events of the whole system under test, and sniffer as observer, it is possible to apply the model in [22] to our problem. However, we note several advantages of our augmented transition based approach. First, we automatically augment the original state machine, which rigorously encodes the protocol specification and is intuitive to design and easy to verify correctness at the same time. Second, we use timed automata to monitor the timing constraints which are common in wireless protocols, while it is known non-trivial to encode such timing information in Hidden Markov Model. Finally, domain knowledge can be exploited in our approach to efficiently prune out unlikely branches during the exhaustive search.

Network Protocol Validation. Lee *et al* [16] studied the problem of passive network testing of network management. The system input/output behavior is only partially observable. However, the uncertainty only lies in missing events in the observation, while in the context of wireless protocol verification, the uncertainty could also be caused by extra events not observed by the tested system. Software model checking techniques [18,10] have also been used to verify network protocols. Our problem is unique because of the observation uncertainty caused by sniffers.

6 Conclusions and Discussions

In this paper, we formally describe the uncertainty problem in validating wireless protocols using sniffers. We propose to systematically augment the protocol state machine to explicitly encode the inherent uncertainty of sniffer traces. We characterize the NP-completeness of the problem and propose practical heuristics to restrict the search to only traces with high likelihood. We implement and evaluate our framework using both NS-3 simulator and real world traces, and show that our framework can tolerance sniffer trace uncertainties and report true violations.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
2. M. Arnold, M. Vechev, and E. Yahav. Qvm: an efficient runtime for detecting defects in deployed systems. In *ACM Sigplan Notices*, volume 43, pages 143–162. ACM, 2008.
3. P. Bahl, R. Chandra, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. Enhancing the security of corporate Wi-Fi networks using DAIR. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 1–14. ACM, 2006.
4. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *FM 2011: Formal Methods*, pages 88–102. Springer, 2011.
5. Y.-C. Cheng, J. Bellardo, P. Benkö, A. C. Snoeren, G. M. Voelker, and S. Savage. *Jigsaw: solving the puzzle of enterprise 802.11 analysis*, volume 36. ACM, 2006.
6. M. Ciabarra. WiFried: iOS 8 WiFi Issue. `https://goo.gl/KtRDqk`.
7. digitalmediaphile. Windows 10 wifi issues with surface pro 3 and surface 3. `http://goo.gl/vBqiEo`.
8. L. Fei and S. P. Midkiff. Artemis: Practical runtime monitoring of applications for execution anomalies. In *ACM SIGPLAN Notices*, volume 41, pages 84–95. ACM, 2006.
9. Gizmodo. The worst bugs in android 5.0 lollipop and how to fix them. `http://goo.gl/akDcvA`.
10. P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.
11. Google. Google contact lens. `https://en.wikipedia.org/wiki/Google_Contact_Lens`.
12. M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Acm Sigplan Notices*, volume 39, pages 156–164. ACM, 2004.
13. A. Kamerman and L. Monteban. Wavelan®-ii: a high-performance wireless lan for the unlicensed band. *Bell Labs technical journal*, 2(3):118–133, 1997.
14. M. Lacage, M. H. Manshaei, and T. Turletti. IEEE 802.11 rate adaptation: a practical approach. In *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 126–134. ACM, 2004.
15. M. Lacage, M. H. Manshaei, and T. Turletti. IEEE 802.11 rate adaptation: a practical approach. *[Research Report] RR-5208*, (¡inria-00070784¿):25, 2004.
16. D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *Network Protocols, 1997. Proceedings., 1997 International Conference on*, pages 113–122. IEEE, 1997.
17. R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Analyzing the mac-level behavior of wireless networks in the wild. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 75–86. ACM, 2006.
18. M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.
19. T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan. Observer effect and measurement bias in performance analysis. 2008.

20. G. F. Riley and T. R. Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, pages 15–34. Springer, 2010.
21. Savvius Inc. Savvius Wi-Fi adapters. `https://goo.gl/l3VXSx`.
22. S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Runtime Verification*, pages 193–207. Springer, 2011.

Appendices

A Proof of Theorem 1

Assume Tr satisfies S^+, and P is a sequence of accepting transitions, we construct a mutation trace Tr' using P and show that Tr' satisfies S.

Initially, let $Tr' = Tr$, then for each *augmented* transition $\langle s_i, s_j, \sigma, g, C' \rangle \in P$:

- If this is a Type-1 transition, add (t, p) to Tr', where t is a timestamp that satisfies g and p is the missing packet.
- If this is a Type-2 transition, remove corresponding (t, p) from Tr'.

It is obvious that Tr' is a mutation trace of Tr, since only receiving packets are removed in the process. Now we show that there exists a accepting transition sequence P' of S^+ on input Tr' that does not contain augmented transitions. In particular, P' can be obtained by substituting all Type-1 transitions with corresponding original transitions, and removing all Type-2 transition. Since P' does not contain augmented transitions, it is also an accepting transition sequence of S on input Tr', hence Tr' satisfies S.

On the other hand, assume $Tr' \in \mathcal{M}(Tr)$ and Tr' satisfies S. Suppose P' is the accepting transition sequences of S on input Tr'. We first note that P' is also the accepting transitions of S^+ on input Tr', since $E \subset E^+$.

We construct a accepting transition sequence P of S^+ on input Tr as follows.

- For each packet $p \in Tr' \setminus Tr$, substitute the transition $\langle s_i, s_j, p, g, C' \rangle$ with the corresponding Type-1 transition $\langle s_i, s_j, \epsilon, g, C' \rangle$.
- For each transition $\langle s_i, s_j, \sigma, g, C' \rangle$ followed by packet $p \in Tr \setminus Tr'$, add a Type-2 self transition $\langle s_j, s_j, p, g, \emptyset \rangle$. This is possible since Tr' is a mutation trace of Tr, thus for all $p \in Tr' \setminus Tr$, $p.src \neq DUT$.

Therefore, Tr satisfies S^+.

B Proof of Lemma 2

To prove this lemma, we use a slightly extended monitor model where boolean variables are allowed but can only be set to `true` or `false` along state transitions. This extension makes the representation of the state machine more succinct yet does not change the complexity of the model, since the boolean variable assignments do not reply on the packet payload.

First, note that the length of mutation trace Tr' is polynomial to the length of Tr because of the discrete time stamp and non-overlapping packets assumption. Therefore, given a state transition sequence as witness, it can be verified in polynomial time whether or not it is an accepting transition sequence, hence both VALIDATION-1 and VALIDATION-2 are in NP.

Next, we show how the SAT problem can be reduced to either one of the two problems. Consider an instance of SAT problem of a propositional formula

F with n variables $x_0, x_1, \ldots, x_{n-1}$, we construct a corresponding protocol and its checker state machine as follows.

The protocol involves two devices: the DUT (transmitter) and the endpoint (receiver). The DUT shall send a series of packets, $pkt_0, pkt_1, \ldots, pkt_{n-1}$. For each pkt_i, if the DUT receives the acknowledgment packet ack_i from the endpoint, it sets boolean variable x_i to be `true`. Otherwise x_i remains to be `false`. After n rounds, the DUT evaluate the formula F using the assignments and sends a special packet, pkt_{true}, if F is `true`. One round of the protocol operation can be completed in polynomial time since any witness of F can be evaluated in polynomial time.

The protocol checker state machine S is shown in Fig. 6. Initially, all x_i is set to `false`. At state s_0, the DUT shall transmit pkt_i within a unit time, transit to s_1 and reset the clock along the transition. At state s_1, either the DUT receives the ack_i packet and set x_i to be `true` ($s_1 \rightarrow s_0$), or the DUT continues to transmit the next packet pkt_{i+1}. After n rounds, the state machine is s_0 or s_1 depending on whether ack_{n-1} is received by the DUT. In either case, the DUT shall evaluate F and transmit pkt_{true} if F is `true`.

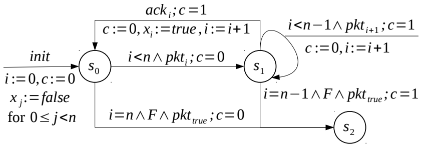

$ack_i; c=1$
$c:=0, x_i:=true, i:=i+1$
$i<n-1 \wedge pkt_{i+1}; c=1$
$c:=0, i:=i+1$
$init$
$i:=0, c:=0$
$x_j:=false$
for $0 \leq j<n$
s_0
$i<n \wedge pkt_i; c=0$
s_1
$i=n-1 \wedge F \wedge pkt_{true}; c=1$
$i=n \wedge F \wedge pkt_{true}; c=0$
s_2

Fig. 6: **Checker State Machine for SAT Problem.**

Consider a sniffer trace $Tr_1 = \{(0, pkt_0), (2, pkt_1), (4, pkt_2), \ldots, (2(n-1), pkt_{n-1}), (2n, pkt_{true})\}$. That is, the sniffer only captures all pkt_i plus the final pkt_{true}, but none of ack_i. It is easy to see that F is satisfiable if S_1^+ accepts Tr_1. In particular, a successful run of S_1^+ on Tr_1 would have to guess, for each pkt_i, whether the Type-1 empty transitions should be used to infer the missing ack_i packet, such that F is *true* at the end. Note that for Tr_1, no Type-2 self transitions can be used since all packets in Tr_1 are sent from the DUT. Therefore, the SAT problem of F can be reduced to the VALIDATION-1 problem of S_1^+ on sniffer trace Tr_1.

On the other hand, consider another sniffer trace $Tr_2 = \{(0, pkt_0), (1, ack_0), (2, pkt_1), (3, ack_1), \ldots, (2n-2, pkt_{n-1}), (2n-1, ack_{n-1}), (2n, pkt_{true}\}$. That is, the sniffer captures all n pair of pkt_i and ack_i packets and the final pkt_{true} packet. Similar to Tr_1, F is satisfiable if S_2^+ accepts Tr_2. A successful transition sequence of S_2^+ on Tr_2 must decide for each ack_i packet, whether Type-2 self transitions should be used, so that F can be evaluated as true at the end. Therefore, the SAT problem of F can also be reduced to the VALIDATION-2 problem of S_2^+ on sniffer trace Tr_2.

Since SAT is known to be NP-complete, both the VALIDATION-1 and the VALIDATION-2 problem are also NP-complete.

Algorithm 2 `pkt_succ` function

```
1: function PKT_SUCC
2:     i := (i+1)%N
3:     succ := succ + 1
4:     count := count + 1
5:     probe := false
6:     if r < $R$ and (succ ≥ $Th_1$ or count ≥ $Th_2$) then
7:         r := r+1
8:         succ := 0
9:         count := 0
10:        probe := true
```

C Detail of ARF Algorithm

Automatic Rate Fallback (ARF) [13] is the first rate control algorithm in literature. In ARF, the sender increases the bit rate after Th_1 number of consecutive successes or Th_2 number of packets with at most one retransmission. The sender decreases bit rate after two consecutive packet failures or if the first packet sent after rate increase (commonly referred as *probing* packet) fails.

Figure 7 shows the state machine S for the packet trace collected at sender (DUT), where $DATA_i^r$ denotes a data packet with sequence number i and bit rate r, $DATA_i^{r\prime}$ is a retransmission packet and Ack is the acknowledgment packet. The `pkg_succ` function is shown in Algorithm 2.

The `succ` variable is used to track the number of consecutive packet successes. It is increased after each packet success , and is reset to 0 after a rate increase or upon a packet failure ($s_1 \rightarrow s_2$). Similarly, `count` is to track the number of packets with at most one retransmission, and is increased after packet success, or for the first packet retransmission ($s_1 \rightarrow s_2$). It is reset when there are two consecutive packet failures ($s_2 \rightarrow s_3$). Finally, the `probe` flag is set upon rate increases to indicate the probing packet, and is cleared upon packet success. The variable `r` is the current bit rate, which is decreased if the probing packet fails ($s_1 \rightarrow s_4$), or every two consecutive failures ($s_2 \rightarrow s_3$). If `r` is not the highest rate, it is increased when either of the two thresholds are reached.

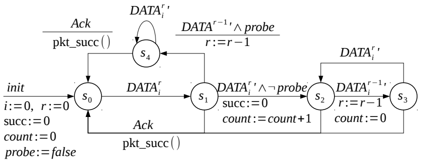


Fig. 7: **Checker State Machine for ARF Rate Control Algorithm.** Timing constraints are omitted for succinctness.

In particular, the bug we found lies in the implementation's `pkt_succ` function in line 6. Instead of checking `count` ≥ `Th_2`, the implementation checks `count == Th_2`.

Note that the `count` variable is incremented twice if a packet succeed after one retransmission: once in $s_1 \rightarrow s_2$, once in the `pkt_succ` function for the retransmission packet. Therefore, if the value of `count` is $Th_2 - 1$ and the next packet succeed after one retransmission, the value of `count` will be $Th_2 + 1$, which would fail the implementation's test of `count == Th_2`.