Wireless Protocol Validation Under Uncertainty

Jinghao Shi[1], Shuvendu K. Lahiri[2], Ranveer Chandra[2], and Geoffrey Challen[1]

[1] University at Buffalo, Buffalo, NY 14120, USA
{jinghaos, challen}@buffalo.edu
[2] Microsoft Research, Redmond, WA 98052, USA
{shuvendu, ranveer}@microsoft.com

Abstract. Runtime validation of 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 the DUT's behavior without instrumentation, but they introduce new validation challenges. Losses caused by wireless propagation mean that sniffers cannot perfectly reconstruct the actual DUT packet trace. As a result, accurate validation 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 monitor state machine, we automatically and completely encode sniffer uncertainty by selectively adding non-deterministic transitions. We characterize the NP-completeness of the resulting decision problem and provide an exhaustive algorithm for searching over all mutated traces, as well as more practical protocol-oblivious heuristics for searching over the most likely mutated traces. We have implemented our framework and show that it can accurately distinguish most 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 Google Iris contact lenses [11], Xbox One wireless controllers [24], and Google Chromecast [23]. 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 [6,9,7].

Runtime validation of the protocol implementations 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 implementation details. This may occur when development has been performed by a third-party that considers the implementation proprietary, or when multiple devices from different vendors are being tested for interoperability. And even when direct instrumentation is possible, the overhead it causes may alter the behavior of the DUT due to the observer effect [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.

Since 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 incomplete. Accurate validation requires distinguishing between these two causes.

We present a new technique enabling validation of protocol implementation using wireless sniffers. Given a monitor state machine representing the protocol being validated, we propose 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. Searching over all the 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:

1. To the best of our knowledge, we are the first to identify the uncertainty problem caused by sniffer in validating wireless protocol implementations.
2. We formalize the problem using a nondeterministic state machine that systematically and completely encodes the uncertainty of the sniffer trace.
3. We characterize the NP-completeness of the validation problem, and present two protocol-oblivious heuristics to prune the search space and make validation possible in practice.
4. We implement the validation framework and evaluate it using NS-3 network simulator [20]. Our framework accurately identifies both introduced and previously unknown violations in NS-3's implementations of the 802.11 protocol.

Due to space limitation, we provide the full proof of lemmas and theorems presented in this paper in the appendices.

2 Background and Motivating Example

We encounter this problem while testing the protocol implementation of a popular game controller. To meet the low latency and low power consumption goals, custom wireless communication protocol was designed. The protocol specification was then handed over to wireless chipset vendors for implementation, which is common industry practice. However, neither implementation details nor trace collection capabilities are provided in the shipped firmware due to Intellectual Property constraints and device resource limitation. Hence using sniffers to validate the protocol implementation is the natural and only option.

The testing has been performed via manual spot-checking of the sniffer trace. We developed a tool to automatically validate certain protocol properties in the sniffer trace, yet often found outrageous amounts of false alarms due to the incompleteness of the sniffer traces, making the tool virtually useless.

To better understand the incompleteness of sniffer trace, consider the IEEE 802.11 (as known as Wi-Fi) transmitter (DUT) state machine shown in Fig. 1. After the DUT sends $DATA_i$—a data packet with sequence number i ($s_0 \to s_1$), it starts a timer and waits for the acknowledgment packet—*Ack*. Either it receives *Ack* within time T_o ($s_1 \to s_0$), or it sends $DATA'_i$—retransmission of $DATA_i$ ($s_1 \to s_2$). Similarly, either it receives the *Ack* within T_o ($s_2 \to s_0$) or aborts transmission and moves on to next packet[3] ($s_2 \to s_1$).

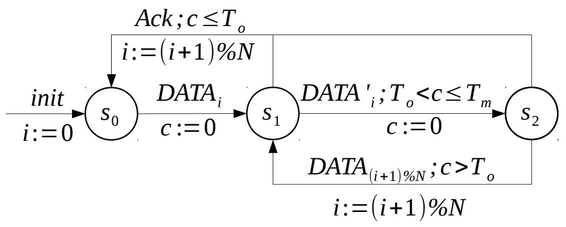


Fig. 1: **Monitor State Machine for 802.11 Transmitter.**

Given the DUT's internal log of packet transmission and reception events, it is trivial to feed such log into the state machine in Fig. 1 and validate the correctness of DUT's protocol implementation. In this paper, however, we assume the DUT implementation is a black box and its internal events are not available. And we seek to validate the DUT implementation using sniffers.

Two fundamental properties in wireless communication bring uncertainty to sniffer's observation: packet loss and physical diversity. The sniffer could either missing packets from the DUT (packet loss), or overhear packets that are sent to but missed by the DUT (physical diversity).

[3] We assume in this example that the DUT will retry at most once to succinctly present the state machine. In practice, multiple retransmissions will be made before aborting.

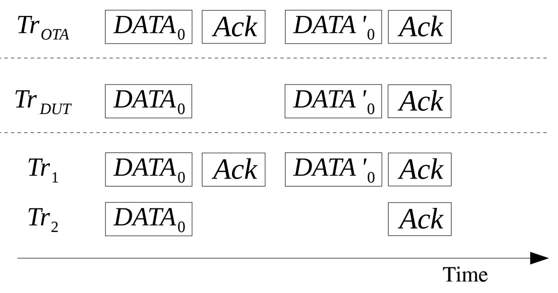


Fig. 2: **Uncertainty of Sniffer Observations.** Tr_{OTA} is the chronological sequence of packets sent by the DUT and the receiver. Tr_{DUT} is DUT's internal events. Tr_1 and Tr_2 are two possible observations of the sniffer.

Consider a correct packet exchange sequence and a few sniffer observations shown in Fig. 2. The DUT first sends $DATA_0$. Suppose the receiver receives $DATA_0$ and sends the *Ack* which the DUT does not receive. Eventually the DUT's timer fires and it sends $DATA'_0$. This time the $DATA'_0$ reaches receiver and the DUT also receives the *Ack*.

In first possible sniffer observation Tr_1 where the sniffer *overhears* the first *Ack* packet, a validation *uncertainty* arises when the sniffer sees the $DATA'_0$: 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 observation 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 *Ack* for $DATA_0$ nor sends $DATA'_0$. Again, without any additional information it is impossible to disambiguate between the sniffer missing certain packets and a bug in DUT's retransmission logic.

Informally, the question we set out to answer in this paper is: given the protocol monitor state machine and the sniffer's observation with inherent uncertainty, how to validate that the DUT behaves as specified?

3 Prerequisites and Problem Statement

3.1 Packet, Trace and Monitor State Machine

The alphabet of the monitor 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. The input of the state machine then corresponds to a 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.

We use *timed automata* [1] to model the expected behaviors of the DUT. 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 refer the readers to [1] for more details about timed automata.

Definition 2. *A protocol monitor state machine S is a 7-tuple $\{\Sigma, \mathbb{S}, \mathbb{X}, s_0, C, E, G\}$, 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.*
- *$\mathbb{X}$ is the set of boolean variables. We use $v = \{x \leftarrow true/false \mid x \in \mathbb{X}\}$ to denote an assignment of the variables. Let $\mathbb{V}$ be the set of such values v.*
- *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 guard conditions defined inductively by*

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

  where $c \in C$ is a clock variable, T is a constant, and x is a variable in $\mathbb{X}$. A transition can choose not to use guard conditions by setting g to be $true$.
- *$E \subseteq \mathbb{S} \times \mathbb{V} \times \mathbb{S} \times \mathbb{V} \times \Sigma \times G \times \mathscr{P}(C)$ gives the set of transitions. $\langle s_i, v_i, s_j, v_j, p, g, C' \rangle \in E$ represents that if the monitor is in state s_i with variable assignments v_i, given the input tuple (t, p) such that the guard g is satisfied, the monitor can transition to a state s_j with variable assignments v_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.

As an example, the monitor state machine illustrated in Fig. 1 can be formally defined as follows:

- $\Sigma = \{DATA_i, DATA'_i, Ack \mid 0 \leq i < N\}$.
- Clock variables $C = \{c\}$. The only clock variable c is used for acknowledgment time out.
- $\mathbb{X} = \{i\}$, as a variable with $log(N) + 1$ bits to count from 0 to N.
- 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.

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.

3.2 Mutation Trace

As shown in the motivation example in Fig. 2, a sniffer trace may either miss packets that are present in DUT trace, or contain extra packets that are missing in DUT trace. Note that in the latter case, those extra packets must be all sent *to* the DUT. This is because it is impossible for the sniffer to overhear packets sent from the DUT that were not actually sent by the DUT.

We formally capture this relationship with the definition of mutation trace.

Definition 4. *A packet trace Tr' is a mutation of sniffer trace Tr w.r.t a DUT if for all $(t, p) \in Tr \setminus Tr'$, $p.dest = DUT$.*

By definition, either $Tr' \supseteq Tr$ (hence $Tr \setminus Tr' = \emptyset$), or those extra packets in Tr but not in Tr' are all sent to the 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 sniffer trace 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 in Appendix A)

3.3 Problem Statement

Lemma 1 shows that $\mathcal{M}(Tr)$ is a *complete* set of guesses of the DUT packet trace. Therefore, the problem of validating DUT implementation given a sniffer trace can be formally defined as follows:

Problem 1. VALIDATION
instance A protocol monitor state machine S and a sniffer trace Tr.
question Does there exist a mutation trace Tr' of Tr that satisfies S?

If the answer is no, a definite violation of the DUT implementation can be claimed. Nevertheless, if the answer is yes, S could still reject Tr_{DUT}. In other words, the conclusion of the validation can either be *definitely wrong* or *probably correct*, but not *definitely correct*. This is the fundamental limitation caused by the uncertainty of sniffer traces.

4 Verification Framework

4.1 Augmented State Machine

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

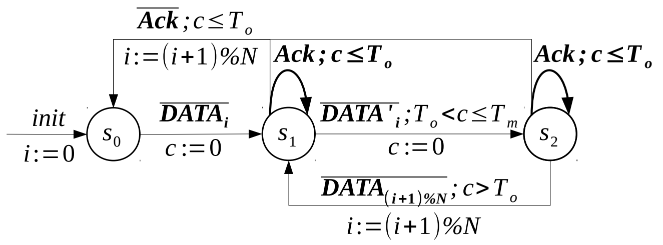


Fig. 3: **Augmented Monitor State Machine.** Augmented transitions are highlighted in bold face. $\overline{Pkt}$ means either ϵ or Pkt.

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. 1. For each existing transition (e.g., $s_0 \to s_1$), 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 $s_1 \to s_0$ and $s_2 \to s_0$, we add a *self transition* with the same trigger packet and clock guards, but empty set of resetting clocks and no assignments to variables. This is to allow the state machine make progress when the sniffer missed such packets.

We make two notes. First, self transitions are added only for packets sent *to* the DUT, since the sniffer will not overhear packets *from* the DUT if they were not sent by the DUT. 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. 2. 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 5. *An augmented state machine S^+ for a monitor state machine S is a 7-tuple $\{\boldsymbol{\Sigma}^+, \mathbb{S}, \mathbb{X}, s_0, C, \boldsymbol{E}^+, G\}$, where $\mathbb{S}, \mathbb{X}, s_0, C, G$ 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 transitions in E.*
- *E_2^+: self transitions (**Type-2**) for transitions triggered by receiving packets.*

Algorithm 1 Obtain Augmented Transitions E^+ from E

```
1: function AUGMENT($E$)
2:     $E^+ := \emptyset$
3:     for all $\langle s_i, v_i, s_j, v_j, p, g, C' \rangle \in E$ do
4:         $E^+ := E^+ \cup \{\langle s_i, v_i, s_j, v_j, p, g, C' \rangle\}$          ▷ Type-0
5:         $E^+ := E^+ \cup \{\langle s_i, v_i, s_j, v_j, \boldsymbol{\epsilon}, g, C' \rangle\}$          ▷ Type-1
6:         if $p.dest = DUT$ then
7:             $E^+ := E^+ \cup \{\langle s_i, v_i, \boldsymbol{s_i}, \boldsymbol{v_i}, p, g, \boldsymbol{\emptyset} \rangle\}$          ▷ Type-2
8:     return $E^+$
```

Algorithm 1 describes the process of transforming E into E^+. In particular, Line 4 adds existing transitions in E to E^+, while line 5 and 7 add Type-1 and Type-2 transitions to E^+ respectively. We have highlighted the elements of the tuple that differ from the underlying Type-0 transition. Note that in Type-2 transitions, both the state and the variables stay the same after the transition.

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.

A successful run of S^+ on sniffer trace Tr yields a mutation trace Tr' which represents one possibility of the DUT trace. Specifically, Tr' can be obtained by adding missing packets indicated by Type-1 transitions to Tr, and removing extra packets indicated by Type-2 transitions from Tr

We show that the VERIFICATION problem is equivalent to the satisfiability problem of Tr on S^+.

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

(Proof in Appendix B)

By Theorem 1, 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.

4.2 Problem Hardness

In this section, we show that the VERIFICATION problem 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 an imaginary sniffer that is able to capture *every* packet ever transmitted, then only Type-2 transitions are needed since the sniffer may still overhear packets sent to the DUT. Similarly, suppose another special sniffer that would not overhear any packets sent to the DUT, then only Type-1 transitions are needed to infer missing packets.

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

Problem 2. VALIDATION-1
Given that $Tr \setminus Tr_{DUT} = \emptyset$ (sniffer does not overhear packets).
instance Checker state machine S and sniffer trace Tr.
question Does S_1^+ accept Tr?

Problem 3. VALIDATION-2
Given that $Tr_{DUT} \subseteq Tr$ (sniffer does not missing packets).
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.*

(Proof in Appendix C)

The hardness statement of the general VALIDATION problem naturally follows Lemma 2.

Theorem 2. *VALIDATION is NP-complete.*

4.3 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 accepting sequence if it exists, thus is exhaustive. In the next sections, we present heuristics to limit the search to accepting sequences of S^+ that require relatively few transitions from $E_1^+ \cup E_2^+$. Due to the NP-completeness of the problem, this also makes the algorithm meaningful in practice.

The main routines of the algorithm is shown in Algorithm 2. 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 S^+'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 6). 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 13) or dropping it (line 26). While in `EXTEND-1`, we guess a missing packet without consuming the next real packet (line 20). Note that since only

Algorithm 2 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 return $nil$[a]
6:     for i $\in$ [0, 1, 2] do
7:         mutation := EXTEND-i(prefix, p::suffix, $s$)
8:         if mutation $\neq$ $nil$ then return mutation
9:     return $nil$
10: function EXTEND-0(prefix, p::suffix, $s$)
11:     for $\langle s, s', p\rangle$[b] $\in E$ do
12:         if suffix = $nil$ then return prefix@p
13:         mutation := EXTEND(prefix@p, suffix, $s'$)
14:         if mutation $\neq$ $nil$ then return mutation
15:     return $nil$
16: function EXTEND-1(prefix, p::suffix, $s$)
17:     for all $\langle s, s', q\rangle \in E_1^+$ do
18:         if q.time > p.time then
19:             continue
20:         mutation := EXTEND(prefix@q, p::suffix, $s'$)
21:         if mutation $\neq$ $nil$ then return mutation
22:     return $nil$
23: function EXTEND-2(prefix, p::suffix, $s$)
24:     for all $\langle s, s, p\rangle \in E_2^+$ do
25:         if suffix = $nil$ then return prefix
26:         mutation := EXTEND(prefix, suffix, $s$)
27:         if mutation $\neq$ $nil$ then return mutation
28:     return $nil$
```

[a] This check should be ignored in the exhaustive algorithm.
[b] $\langle s, s', p\rangle$ is short for $\langle s, *, s', *, p, *, *\rangle$

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 12 and line 25).

It is not hard to see that Algorithm 2 terminates on any sniffer traces: each node in the transition tree only has finite number of possible next steps, and the depth of Type-1 transitions are limited by the time available before the next packet (line 18).

4.4 Pruning Heuristics

In the face of uncertainty between a possible protocol violation and sniffer imperfection, augmented transitions provide the ability to blame the latter. The exhaustive nature of Algorithm 2 means that it always tries to blame sniffer imperfection whenever possible, making it reluctant 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 strictness of the likelihood constraint represents a trade-off between precision and recall of validation: the more strict the constraints are, the more false positive violations will potentially be reported, hence the lower the precision yet higher recall. 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.

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. Next, we propose two *protocol oblivious* heuristics based on the sniffer loss probabilities and general protocol operations. Both heuristic contains parameters that can be fine tuned in practice.

***NumMissing*(d, l, k)** This heuristic 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 size 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 this heuristic is that the sniffer's empirical packet loss probability can usually be measured before validation. Therefore, the likelihood that the sniffer misses more packets that prior measured loss ratio is quite low. The value of l and k can then be configured such that k/l is marginally larger than the measured ratio.

***GoBack*(k)** This heuristic states that the search should only backtrack 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. Note that we do not require the protocol to specify an exact transaction boundary, but only need k to be sufficiently large to cover a transaction.

5 Case Studies

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

5.1 802.11 Data Transmission

In this section, we first show that our verification framework can improve verification precision by inferring the missing or extra packets using the augmented transition framework. We then demonstrate the ability of our framework to detect true violations by manually introducing bugs in the NS-3 implementation and show the precision and recall of validation results.

Experimental Setup We set up two Wi-Fi devices acting as the transmitter (DUT) and receiver respectively. Another Wi-Fi device is configured in monitor mode and acts as the sniffer. During the experiments,we collect both the DUT packet trace (the ground truth) and the sniffer trace.

Verifying Unmodified Implementation In the original monitor state machine shown in Fig. 1, we set acknowledgment timeout $T_o = 334\mu$s, maximum retransmission delay $T_m = 15$ms according to the protocol. We also adapt the state machine to include multiple retransmissions[4] instead of one.

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. Pr_{ed} represents the characteristics of the system being tested, while Pr_{ds} and Pr_{es} represent the sniffer's quality in capturing packets.

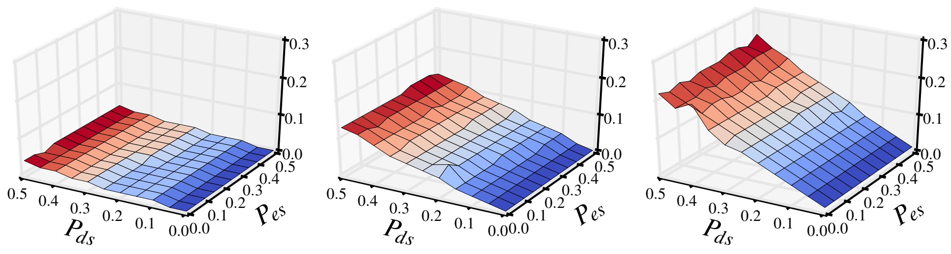
We vary each of the three probability, Pr_{ds}, Pr_{es} and Pr_{ed}, from 0 to 0.5 (both inclusive) with 0.05 step. For each loss ratio combination, we ran the experiment 5 times, and each run lasted 30 seconds. In total, 6655 ($11^3 \times 5$) pair of DUT and sniffer packet traces were collected.

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 *GoBack*(k) heuristic, we set $k = 7$, which is the maximum number of transmissions of a single packet. For the *NumMissing*(d, l, k) heuristic, we set the sliding window size $l = 100$, and $k = 80$ such that no violation is reported. The relationship of k and validation precision is studied in next section.

Next, we present detailed analysis of the augmented transitions on the sniffer traces. The goal is to study for a given system packet loss probability Pr_{ed}, how the sniffer packet loss properties (Pr_{ds} and Pr_{es}) affect the difference between

[4] The exact number of retransmissions is not part of the protocol, and NS-3 implementation set this to be 7.



(a) $0.05 \leq Pr_{ed} \leq 0.15$ (b) $0.2 \leq Pr_{ed} \leq 0.35$ (c) $0.4 \leq Pr_{ed} \leq 0.5$

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

the DUT trace and the mutation trace, which represents a guess of the DUT trace by the augmented state machine based on the sniffer 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.

Fig. 4 shows the Jaccard Distance between mutation and its corresponding DUT trace. We make the following observations. First, for a given system loss probability Pr_{ed} (each sub-figure), the lower the sniffer packet loss probability (Pr_{ds} and Pr_{es}), the smaller Jaccard distance between the DUT and mutation trace. Intuitively, this means a sniffer that misses less packets can enable our framework to better reconstruct the DUT trace.

Second, we observe a *protocol-specific* trend that Pr_{ds} is more dominant than Pr_{es}. This is because retransmission packets of the same sequence number are identical, hence when the sniffer misses multiple retransmission packets, our framework only needs to infer one retransmission packet to continue state machine execution.

Finally, as the system loss probability Pr_{ed} increases, the Jaccard distance increases more rapidly as Pr_{ds} increases. This is because the ratio of retransmission packet increases along with Pr_{ed}.

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:

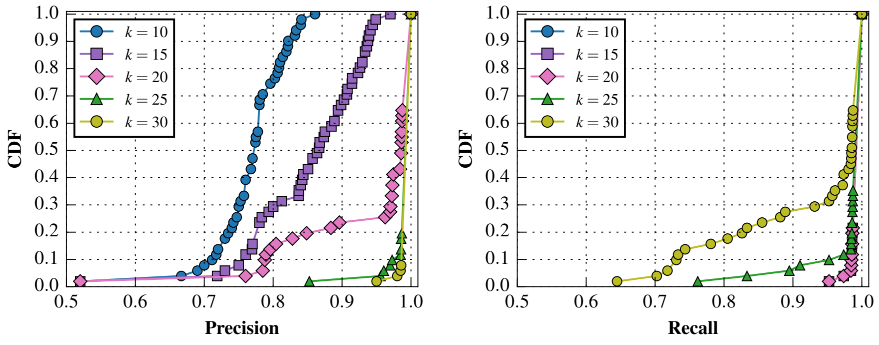


Fig. 5: **Precision and Recall of Validation Results.**

- **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. We choose one type of such bugs in each run.
- **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 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.

Fig. 5 shows the CDF of precision and recall of the 51 experiments for various k values. For precision, as expected, the more tolerant the search to sniffer losses (larger k), the more tolerant the framework is to sniffer losses, and 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.

5.2 ARF Rate Control Algorithm

We report a bug found in NS-3 ARF [13] implementation which causes the sender gets stuck at a lower rate even after enough number of consecutive successes. The bug was detected using the sniffer trace and confirmed using the DUT trace and source code inspection. The details of this bug can be found in Appendix D.

6 Related Works

Hidden Markov Model (HMM) Approach. When considering the whole system under test (both DUT and endpoint), the sniffer only captures a subset of the all the packets (events). This is similar to the event sampling problem in runtime verification [4,12,2,8]. Stoller *et al* [22] used HMM-based state estimation techniques to calculate the confidence that the temporal property is satisfied in the presence of gaps in observation.

While it seems possible to adapt the method in [22] to our problem, we note several advantages of our augmented monitor and prioritized search procedure. First, the automatically augmented state machine precisely encodes the protocol specification and the uncertainty; this is intuitive to design and natural for reporting the evidence for a trace being successful. We do not require a user to specify the number of states of underlying HMM, or accurately provide underlying probabilities. Second, we use timed automata to monitor the timing constraints which are common in wireless protocols; it may be non-trivial to encode such timing information in HMM. Finally, we can exploit domain knowledge to devise effective pruning heuristics to rule out unlikely sequences 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. However, they do not provide any formal guarantees even for cases when we report a definite bug. 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.

7 Conclusions

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 uncertainty of sniffer traces. We characterize the NP-completeness of the problem and propose both exhaustive searching algorithm and heuristics to restrict the search to only probable traces. We present two case studies using NS-3 network simulator to demonstrate how our framework can improve validation precision and detected real bugs. We plan to apply our framework in testing the game controllers and other real world protocols.

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.
23. Wikipedia. Chromecast. `https://en.wikipedia.org/wiki/Chromecast`.
24. Wikipedia. Xbox One controller. `https://en.wikipedia.org/wiki/Xbox_One_Controller`.

Appendices

A Proof of Lemma 1

Let $\Delta = Tr \setminus Tr_{DUT}$ 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 4, $Tr_{DUT} \in \mathcal{M}(Tr)$.

B 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 $\langle s_i, v_i, s_j, v_j, \sigma, g, C' \rangle \in P$:

– 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 $\langle s_i, v_i, s_j, v_j, p, g, C' \rangle$ with the corresponding Type-1 transition $\langle s_i, v_i, s_j, v_j, \epsilon, g, C' \rangle$.
– For each transition $\langle s_i, v_i, s_j, v_j, \sigma, g, C' \rangle$ followed by packet $p \in Tr \setminus Tr'$, add a Type-2 self transition $\langle s_j, v_j, s_j, v_j, p, g, \emptyset \rangle$. 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^+.

C Proof of Lemma 2

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 monitor 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.

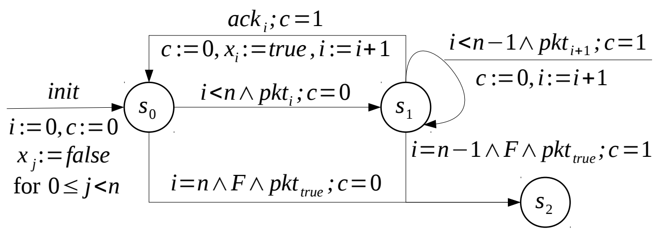

$ack_i; c=1$
$c:=0, x_i:=true, i:=i+1$
$i<n-1 \wedge pkt_{i+1}; c=1$
$c:=0, i:=i+1$
$init$
$i:=0, c:=0$
$x_j:=false$
for $0 \le j<n$
s_0
$i<n \wedge pkt_i; c=0$
s_1
$i=n-1 \wedge F \wedge pkt_{true}; c=1$
$i=n \wedge F \wedge pkt_{true}; c=0$
s_2

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

The protocol monitor 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` ($s_1 \to s_0$), 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`.

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.

D 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.

Fig. 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 3.

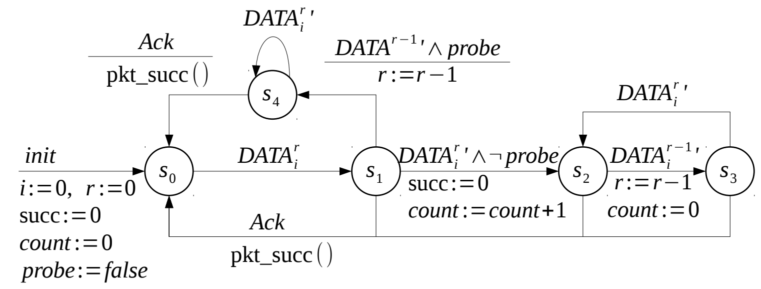


Fig. 7: **Monitor State Machine for ARF Rate Control Algorithm.** Timing constraints are omitted for succinctness.

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.

Algorithm 3 `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 $\geq Th_1$ or count $\geq Th_2$) then
 7:         r := r+1
 8:         succ := 0
 9:         count := 0
10:         probe := true
```

In particular, the bug we found lies in the implementation's `pkt_succ` function in line 6. Instead of checking `count` $\geq$ `Th_2`, the implementation checks `count == Th_2`. This bug also exists in the NS-3 implementation of Adaptive ARF (AARF) algorithm [14] and the pseudo code implementation of AARF in [15].

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`.