Extending Automated Protocol State Learning for the 802.11 4-Way Handshake

Chris McMahon Stone[1], Tom Chothia[1], Joeri de Ruiter[2]

[1] School of Computer Science, University of Birmingham, United Kingdom
[2] Radboud University, Nijmegen, The Netherlands

Abstract. We show how state machine learning can be extended to handle time out behaviour and unreliable communication mediums. This enables us to carry out the first fully automated analysis of 802.11 4-Way Handshake implementations. We develop a tool that uses our learning method and apply this to 7 widely used Wi-Fi routers, finding 3 new security critical vulnerabilities: two distinct downgrade attacks and one router that can be made to leak some encrypted data to an attacker before authentication.

1 Introduction

Automated, systematic analysis of protocol implementations has proven to be an effective tool for security analysis, approaches taken include fuzz testing [5,7], model-based testing [6,32] and protocol state fuzzing (also known as state machine inference) [1,11,26]. The latter of these methods works by learning the state machine implemented by a particular device or application, in a black-box fashion, by sending different sequences of messages and observing the corresponding outputs. Analysis of these state machines can then be carried out to spot any unexpected logic flow. Such discoveries could be benign divergences from the protocol specification, or result in security vulnerabilities.

In this paper we utilise state machine inference in order to carry out a black-box analysis of implementations of the IEEE 802.11 4-Way Handshake protocol. This widely used protocol is the means by which authentication and session key establishment is carried out on IEEE 802.11 (WPA or WPA2 certified Wi-Fi) networks. In contrast to the manual, model-based testing of the 4-Way Handshake by Vanhoef et. al [32], our method has the advantage of being fully automatic. Manual analysis is a long and arduous task, and requires extensive knowledge of the protocol specification to decide whether every possible test case should fail or pass. Automated learning only requires the tester to specify a set of the possible input messages, i.e., the generation of tests is fully automatic and complete. Furthermore, state machine learning automatically adapts future and successive test cases according to the results of previous ones. For example, if one particular message sequence discovers some erroneous state or unexpected output, it does not stop testing there. The algorithm will continue to explore the state space beyond this and therefore cover more ground than is possible with model-based testing.

A naive application of learning to the 4-Way Handshake protocol would fail to handle the implementations time-based behaviour, e.g., message retransmissions and timeouts, though, in general, time-based behaviour can be entirely arbitrary. In protocol settings, past studies have needed to artificially suppress time-based behaviour, as formal time learning algorithms are non-practical due to their high complexity (see for example [13]). This has been done in various ways, for instance, ignoring re-transmissions and manually setting timeouts for responses to ensure time behaviour is not triggered [10,11], or mapping multiple outputs within manually specified times to single state transitions [26,28]. The former technique disables time learning altogether. In the latter, timeouts are manually identified and multiple responses are merged into one, reducing the state space but potentially missing important behaviour.

The quality of the transmission medium and query interfaces can also effect the ability to learn a system. Sometimes a response might be missed and incorrectly marked as a timeout, or a query is not processed by the target and a retransmission occurs, effectively making the system non-deterministic. This poses an issue for naive model-based learning, which requires that the system under test is completely deterministic.

In this paper we propose practical methods to efficiently learn protocol time behaviour and overcome non-determinism. To learn time behaviour we reduce the complexity by making reasonable assumptions about the operation of network protocols. We separate time learning into a secondary learning step. This enables us to first learn non-time based behaviour, without incurring the costly time-complexity that timeouts induce. Throughout this process, we run an error correction method that handles query-response inconsistencies, thereby ensuring learning termination. We implement these methods and use our tool to learn models of the 4-Way Handshake on 7 access points, without which would not have been possible. Our results include the discovery of three vulnerabilities: two distinct downgrade attacks and leakage of multicast data. To summarise, our contributions are as follows:

- We adapt standard Mealy machine inference to learn common time based behaviour in protocols. This is done efficiently and without the need for complex timed automata modelling.
- We provide a practical method to overcome occasional non-deterministic behaviour in protocols.
- We implement our solution and carry out protocol state fuzzing of a range of 4-Way Handshake implementations.

Our tool, along with model diagrams and other information related to this work will be made available online[3].

2 Related Work

State Machine Learning methods, particularly those based on LearnLib library [25,17], have been successfully applied to demystify legacy software [20]

[3] https://chrismcmstone.github.io/wifi-learner/

and combined with fuzzing for software deobfuscation [18]. The technique has also been used in security related use cases, such as TLS [26], SSH implementations [11], the biometric passport [2] and bank cards [1].

Not in a security setting, but still relevant due to their handling of timing and retransmissions, Fiterău-Broştean et. al [10] carry out a combined model learning and model checking of TCP implementations. Due to the lack of expressivity of Mealy machines, they eliminated the timing based behaviour and retransmissions. To achieve this, they make sure the learning queries were short enough to not trigger any timed behaviour and ensured that the network adapter ignored all retransmissions. Similarly, in a study involving the application of active learning to IoT communication, Tappler et al. [28] deal with timeouts by adopting the technique used by [26], whereby a manually learned single timeout is set for the receipt of all messages to all queries. All messages received within that time are then mapped to an abstract output symbol. The problem with this approach is that it does not allow queries that are interleaved between consecutive message responses. It also assumes the timeouts are the same for all queries. Our approach on the other hand only requires an upper-bound timeouts and learns time related states such that querying is permitted providing the responses are non-retransmissions. Jonsson et al. [19] have presented some preliminary work on the theoretical side of learning Mealy machines with timers however this work has not yet lead to a practical implementation.

IEEE 802.11, also commonly referred to as Wi-Fi, has been the subject of a wide array of past research. The original Wi-Fi security mechanism, WEP, is broken [12,29]. WEP was replaced by TKIP (based on the RC4 cipher) and then CCMP (based on AES). While TKIP is insecure it is still supported by most WPA2 access points (APs). The 4-Way Handshake, which is deployed to authenticate clients and negotiate session keys, has undergone extensive formal analysis [15,22,16,33]. Denial of Service vulnerabilities were discovered [15,22], and fixes [16] integrated into the 802.11i specification. The design of the 4-Way Handshake was analysed by Vanhoef et al. [30], who focused on the transmission of the group-key, for which a downgrade attack was discovered that forces the group key to be encrypted using the vulnerable RC4 cipher.

The security of Wi-Fi implementations has also been the subject of many studies [7,21]. Vanhoef et al. apply manual, model-based testing techniques [32], which resulted in the discovery of different DoS and downgrade attacks. More recently, Vanhoef et al. discovered a series of vulnerabilities in how retransmissions of key exchange messages are handled, which lead to the reuse of keystreams [31].

3 Background

3.1 The 802.11 4-Way Handshake

The full 802.11 4-Way Handshake consists of a network discovery and a 802.11 authentication and association stage:

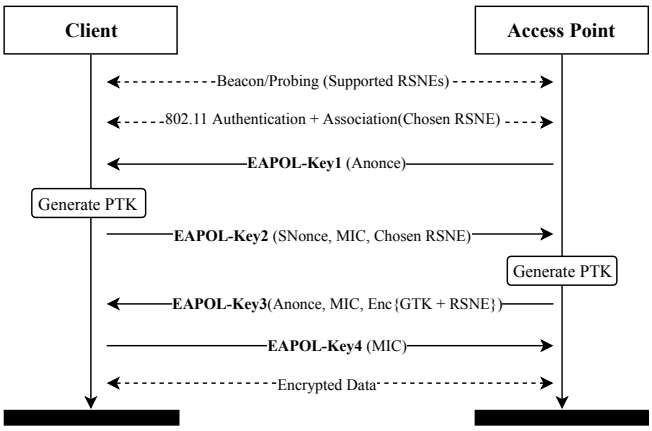


Fig. 1: The 4-Way Handshake.

Network Discovery This stage consists of the stations (clients) searching for available networks and their capabilities. This is done passively, by observing broadcasted Beacons, or actively, by sending and receiving probes. The stations learn which cipher suites are supported (TKIP and/or CCMP) and which version of WPA (1 or 2). Both the cipher suites and WPA version are encapsulated in the Robust Security Network Element (RSNE).

Authentication and Association Before the 4-Way Handshake, the client must "authenticate" and associate with the AP. Here "authentication" is simply an exchange of messages that any client can carry out. The real authentication takes place in the 4-Way Handshake. In the association stage, the client chooses an RSNE and the AP will subsequently accept or reject the connection based on that choice. If accepted, the 4-Way Handshake will then begin.

The 4-Way Handshake provides mutual authentication for a client and authenticator (usually an access point) based on a pre-shared key (PSK). The PSK is used in combination with two nonces, a client nonce (SNonce) and authenticator nonce (ANonce), as well as the MAC addresses of both parties, to generate a session key: the Pairwise Transient Key (PTK).

The 4-Way Handshake, as shown in Figure 1, is initiated by the AP, who communicates its nonce to the client. The client then generates its own nonce, and sends it to the AP in Message 2, along with a Message Integrity Code (MIC) that is calculated over the whole frame using the PTK. The AP can then verify the client has derived the correct PTK by generating the PTK itself, and checking that the MIC is valid. It can also detect a downgrade attack by verifying that the RSNE matches that in the earlier Association stage. If all is well, the AP responds with Message 3, which contains the encrypted Group Key and RSNE. The client can then verify the RSNE is consistent with previous messages, if so acknowledge with Message 4 and if not, abort the connection.

Messages are encapsulated within EAPOL-Key frames. These include nonces, version numbers, MICs, replay counters and so on. In our state machine learning of the 4-Way Handshake we only consider the most crucial of these (with respect

to security). Our chosen fields are specified in Section 4.5. The reader can find complete information on EAPOL-Key frame structure and contents by referring to the 802.11 specification [14].

3.2 State Machine Learning

We use Mealy machines to formalise the state machines that are implemented for the 4-Way Handshake.

Definition 1. *A Mealy machine is a tuple* $(I, O, Q, q_0, \delta, \lambda)$*, where* I *and* O *are the sets of input and output symbols (also known as input and output alphabet respectively),* Q *is the non-empty set of states,* $q_0 \in Q$ *is the initial state,* δ *is a transition function* $Q \times I \to Q$*, and* λ *is an output function* $Q \times I \to O$*.*

When a Mealy machine is in a state $q \in Q$ and receives as input $i \in I$, it transitions to the state $\delta(q, i)$ and produces an output $\lambda(q, i)$.

In the context of learning protocol implementations, we consider Mealy machines that are *complete* and *deterministic*. This means that for each state $q \in Q$, and input $i \in I$, there is exactly one mapping specified by δ and λ.

A classical procedure for learning a state machine is using the L^* algorithm by Angluin [4]. This approach was adapted by Niese to learn Mealy machines [23] and later optimized by Shabaz et al. [27]. The approach consists of two components: An *oracle* (or *teacher*), that acts as an interface to the executing SUL (System Under Learning), and a *learner*, that is only aware of the input and output symbol sets I and O, and can additionally request the oracle to reset the SUL to the start state q_0.

The algorithm works by sending *output queries* that are strings from I^+. The oracle responds with the corresponding output strings from the machine. Each *output query* is preceded by a *reset query*. Using the responses the learner builds up a hypothesis of the state machine as implemented in the SUL.

The next stage of the algorithm is to send an equivalence query to the *oracle*. In this stage, the hypothesis is checked against the actual state machine. If the *oracle* states that the hypothesis is correct, the algorithm terminates. Otherwise, the oracle will respond with the contradicting output string, i.e. a *counterexample*. In the latter case, the learner refines its hypothesis with the *counterexample* and continues the learning process until it has a new acceptable hypothesis. Note that as this is black-box testing, i.e. the oracle cannot access the internal implementation of the SUL, and only a finite number of test cases can be performed, the equivalence checking can only be approximated. The consequence of this is that in some cases it may only be possible to learn a subset of the SUL's behaviour. The most popular learning algorithms are implemented in the LearnLib[24] library, which we use in the development of our tool.

4 Adapting State Machine Learning For Wi-Fi

4.1 Learning Protocols with Errors

A requirement of existing state machine learning methods is that the SUL behaves in a totally deterministic manner, i.e., the same message sent to the device always leads to the same reply. While protocols such as the 4-Way Handshake are specified as deterministic, in practice, the unreliable medium will occasionally lead to lost and corrupting packets and so not meet this requirement. Therefore, to be able to learn these implementations, we must provide a method which stops occasional errors disrupting the learning process.

Running our algorithm on 7 routers, LearnLib reported non-determinism for between 0.5% and 8% of queries (full details are in Section 5). This error rate means that most attempts to learn a router will fail before the state machine can be found. The errors were mainly due to either a message not being received and the response timing out, or a message not being received and a previous message being retransmitted. In the later case, there is no way to tell from a single response alone if the message is a genuine reply to a query or if it is a retransmission due to a lost message.

To deal with non-determinism we maintain a record (or cache), separate from LearnLib, which records all input sequences, all corresponding responses, and the number of times those inputs and responses have been seen. LearnLib will throw an exception when a series of inputs gives a different output to one we have previously seen. We can then handle this exception, and execute a form of 'majority vote' error correction in order to decide on the correct response. This works as follows:

1. Whenever we execute a query (and for each subsection of the query) we record the query, and the response seen.
2. When LearnLib reports non-determinism we record the query and observed response (which could be a timeout) and we look at the total observations for all responses to the query that triggered the exception. Then:
   (a) If the response that triggered the exception is now the strictly most common response, we decide that our previous observations must have been errors. We then remove all queries which have the prefix that triggered the exception from our database of learnt queries, because we concluded they were based on learning an error.
   (b) If the response that triggered the exception is not the strictly most common response, we decide that the response seen is an error, and we retry the query (after updating our record of seen responses).

To avoid non-determinism in equivalence queries, we take a more straightforward approach. If a counter-example is found, then it is repeatedly queried against the SUL, with varying time gaps in-between. Only if the results are consistent is the counter-example then processed by the learning algorithm.

On average, we require in the region of 1000 queries to learn a model. Our method, and optimisations, leads to queries being executed an average of 15

times. Assuming the highest error rate we saw of 8% means that the chance that we learn an error response, rather than the correct response for any query is less than 0.01% (full calculations are given in the appendix). Working backwards from the failure probability, we find that our method will have a 95% confidence of returning the correct automata for error rates of up to 10%. Higher confidence and higher acceptable error rates can be achieved by retrying queries that are not strictly needed by our method, e.g., if we repeat queries to ensure that they are tried at least 100 times we can provide 95% confidence of learning an automata correctly for error rates of up to 30%.

When an error response becomes the most common response to a particular query our method will discard useful information and so be inefficient. For the worst error rate we observed, 8% we calculate the probability of discarding a correct response to a query with 15 tests as 0.00756, more tests do not increase this probability significantly. On the other hand an error rate of 30% would lead to a 0.18 probability of discarding useful data. We note that for such high error rates we could cache the learn queries rather than discarding them so as to avoid having to relearn responses.

4.2 Learning Time-based Behaviour

To accommodate time behaviour into our models, we first make a number of assumptions about the types of time-based behaviour we expect from protocols like the 802.11 handshake. These assumptions include the types of timers in operation and what we consider to be a change of state. This allows us to enforce restrictions on the types of queries that can be executed, thereby making the problem of learning timed models tractable.

Assumption 1. At any given state, there is only one timer in operation, which could expire and trigger output.

To achieve feasible learning times within the Mealy machine model, we limit the number of timers so that there is never more than one timer running at the same time. Indeed, for the purpose of learning the 802.11 handshake this was sufficient. We believe this also to be the case for other similar protocols.

Assumption 2. If a message is retransmitted, it is only when these retransmissions stop, that the state of the SUL will change.

What we mean by this, is that in the scenario of the SUL retransmitting a message, the only aspect of the state that has changed, is the progression of time. Conversely, if a transmitted message differs from the previous transmitted message, then we infer that the state of the SUL has changed. It is not likely that the SUL will retransmit messages indefinitely. Most protocols will implement some sort of timeout mechanism as we will see.

It follows from Assumption 2 that we can consider a retransmission state as a sub-state of its parent. That is, since it is only time that has progressed, all query-responses will remain the same, therefore:

Assumption 3. Any queries after, a observing a retransmitted message, will have the same responses as before the retransmission.

Additionally, we assume that the modeller is able to provide estimated values for a normal response time and upper-bound timeout. The normal response time should be large enough to give the SUL sufficient time to provide non-timer based responses. Essentially, as long as it takes the SUL to receive and process a message, and send a response. In Wi-Fi we set this in the region of 0.2 - 0.5 seconds. For other protocols, or testing set ups, the value should be set according to the quality of the medium on which the protocol is running. For example, one could conceive of a protocol running across further distances, and as such require a longer time allowance for single input/output queries. The second value is an upper-bound timeout. This is required to prevent the learner waiting endlessly if there has been a silent timeout. It should be sensibly set to a maximum value that you expect the SUL to maintain a connection for. E.g., we set this to 20 seconds, as we expect any timers to have expired and connections to be dropped if the handshake has not completed within that time.

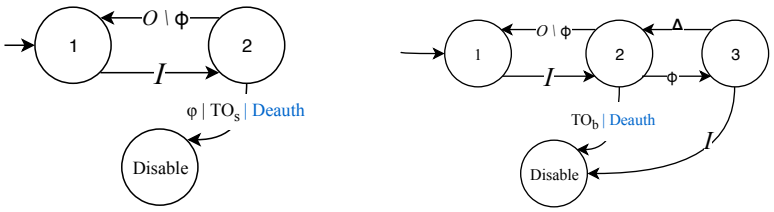
Solution Overview

In our solution, we split the learning procedure into two stages. The first stage will discover behaviour such as the normal flow of the handshake, and states unrelated to time. I.e., we first build a *base* model, which we can later use to learn time behaviour. This way, we can carry out extensive and thorough equivalence checking of the base model, without triggering long timeouts - which causes a blow up in learning time-complexity. The latter stage then uses the base model to identify time-based states, including retransmissions, timeouts and anything else. To this end, we employ two measures. First we use the cache described in Section 4.1, which records all query/responses in a database. This enables us to separate each stage of the learning. Second, we adopt the I/O automata learning method presented in [3]. That is, we employ a transducer that translates the non-Mealy-machine compatible SUL behaviour, into sequences of query-responses that the Mealy machine can understand. This technique enables us to enforce learning restrictions for each corresponding stage. The transducer is implemented as a state machine itself, namely a learning purpose (LP). We construct the learning purpose such that it enforces the following restrictions on the types of permitted queries.

1. Each input symbol $i \in I$ constituting a query, maps to a single output from the set $O \cup \{TO_s \vee TO_b\}$. Where, TO represents a timeout, which is set to the normal response time for the first stage (TO_s), and to the upper-bound timeout for the second (TO_b).
2. If a retransmission[4] is observed, we disable all inputs. An exception is made in the second learning stage where we allow the delay action Δ beyond this point.

The learning purpose representing the described properties for each corresponding stage is depicted in Figure 2. We can see that the learner will begin

[4] Retransmissions definitions can be customised. For the purpose of testing Wi-Fi, we define a retransmission to be an identical message as before, with the exception of the Replay Counter value.



(a) Stage one (b) Stage two

Fig. 2: Learning purposes used for each learning stage. The two timeouts are indicated by TO_s, TO_b, ϕ indicates the last accepted response (retransmission), and Δ a delay action.

at state 1, where any input is enabled. From there, the resulting output O from the SUL will determine the next transition and so on. As soon as the *disable* state is transitioned to, any subsequent inputs will be disabled, meaning that corresponding outputs will be the '$-$' symbol. We include an optimisation of this feature which is detailed in Section 4.4.

When testing 802.11 handshake implementations, we can make some adjustments to the learning purpose to improve efficiency further. Since we know that a `Deauth` indicates a reset of the protocol, we can disable any queries which trigger this output. This modification is highlighted in blue in Figure 2.

Stage 1 Learning Run learning with the full alphabet and the learning purpose from Figure 2a enforced. Once a hypothesis has been produced, we run Chow's W-Method [9] for equivalence checking. This guarantees all states within the restriction of the learning purpose will be discovered (given an upper bound on the number of states). On average this stage will complete quickly, as all time based behaviour (which dramatically increases the execution time of each query) is ignored. Any counter-examples discovered in this stage will be recorded in order to reconstruct the model in the second stage.

Stage 2 Learning Given the base model we learn in the first stage, we can then begin to learn the time related behaviour as follows:

1. Firstly, we delete all entries in the cache oracle that have resulted in the small timeout - TO_s. When learning is restarted, these deleted entries will be posed to the SUL again, this time with the new learning purpose from Figure 2b.
2. Learning is restarted using the new learning purpose. Each query in this stage will first check the cache oracle to see if there is a corresponding response from the first stage. Once a hypothesis has been conjectured, we apply the same counterexamples learned in stage 1.
3. We then begin an equivalence checking stage with the intention of learning all timeouts. That is, for each state already learned, we simply pose queries consisting of many delay actions. The resulting model will represent the base model from stage one, with time based behaviour included. Any non-retransmission, timeout or disconnect states disovered in this stage will also undergo further equivalence checking.

4.3 Broadcast/Multicast Traffic

In addition to unicast traffic, 802.11 networks must facilitate the transmission of messages via broadcast or multicast distribution. The former, broadcast addressing, is where messages are sent to all nodes on the network. The latter, multicast addressing, is another form of one-to-many distribution where messages are sent to a select subset of nodes on the network. The existence of these types of messages on a network poses a problem for learning a deterministic state machine exhibited by an AP. The reason for this is that the processes producing this traffic on the network are generally independent of that running the 4-Way Handshake. Moreover, other nodes on the network can produce this traffic.

One solution to avoid this source of non-determinism would be to ignore these messages. However, this is not an option if we want to incorporate this traffic into our state machine model. Instead, we make a fundamental assumption about what exactly indicates a state change: we assume that multicast or broadcast message will never indicate a state change. In the context of Wi-Fi, this makes sense—the 4-Way Handshake is between the AP and an individual client, as such, all indications of this protocol state change will be made with unicast messages. Working under this assumption we are able to incorporate broadcast/multicast message observation into our model as follows:

1. Learn the model as defined in previous sections but ignore all broadcast/-multicast messages.
2. We then transition to each of the states, and wait for a fixed period, with the intention of detecting any broadcast/multicast traffic. This information is then integrated into the model.

4.4 Additional optimisation

Query Disabling The constraints that we enforce with the learning purpose (see Figure 2), such as disabling any queries beyond a deauthentication or timeout, can be exploited for further efficiency gain. Namely, if we ever observe a query response containing an 'disable output' ($-$), then we know that any aditional inputs beyond that point will also have the 'disable output'. This enables us to maintain a cache of all queries, and their corresponding responses, that result in the learning purpose transitioning to the disable state. This cache can then be used as a lookahead oracle for further queries. For example, say the query $q = \{assoc, delay, data, data\}$ results in response $r = \{accept, E1, timeout, -\}$. The lookahead oracle can then record this query-response pair, as it ended up in a disable state (indicated by the fourth output $-$). If then, the learner poses the query $q_2 = \{assoc, delay, data, data, E4\}$, we already know what the response will be because q is a prefix of q_2, and q ended up in the disable state. Therefore, we can automatically generate the response $r_2 = \{accept, E1, timeout, -, -\}$ without actually querying the SUL.

WPA/2 Specific optimisation In Section 4.2, we show how exploiting our prior knowledge of observing the *Deauthentication* frame, to indicate a reset of

Parameter	Tag	Values	Description
Key Descriptor	KD	WPA1/2, WPA2, RAND	Indicates the EAPOL Key type: WPA, WPA2 or a random value.
Cipher Suites	CS	MD5, SHA1	Ciphers and hash functions used for encrypting the Key Data field and calculating the MIC. Options are MD5+RC4 or SHA1+AES.
RSN Element	RSNE	cc, tc, ct, tt	The chosen ciphersuite combination of TKIP (t) and CCMP (c) for the group and unicast traffic respectively.
Key Information	KF	P, M, S, E	The combination of four flags in the Key Info field: Pairwise (P), MIC (M), Secure (S), Encrypted (E), or - when none is set.
Nonce	NONC	W	The Nonce field contains a consistent (default) or inconsistent (W) nonce.
MIC	MIC	F	The MIC field contains a valid (default) or invalid (F) Message Integrity Code.
Replay Counter	RC	W	The Replay Counter is set to a correct (default) or an incorrect value (W).

Table 1: Parameter definitions for the 802.11 handshake input alphabet.

the protocol, can be used to improve learning efficiency. Similarly, we also implement a check which disables queries after a successful handshake/connection has completed and verified.

4.5 4-Way Handshake Input/Output Learning Alphabet

Inputs Our abstract input alphabet consists of messages of the structure:

$$i \in I := \texttt{MsgType(Params)}$$

Where `MsgType` is one of {`Association, EAPOL 2, EAPOL 4`} and has associated parameters defined in the table below. Associations only permit the RSNE parameter, whereas for EAPOL-Key messages, it can be any. We also include the Delay action (Δ), (Unencrypted) Data, and Encrypted Data (TKIP and AES). We denote the Broadcast/Multicast Delay input (described in Section 4.3) as BRD in our models. In total our input alphabet consists of 45 unique messages. We note that a complete set of all possible combinations of the various EAPOL fields would consist of 1000s of frames. We therefore select the most important fields and values with respect to security.

Outputs Messages received as output from the AP are parsed into the following format:

$$o \in O := \texttt{MsgType(Params)|Timestamp}$$

Where `Timestamp` indicates the time elapsed since the last received message.

4.6 Implementation Details

Network Data When learning the state machines of our selected APs, we ensure that there is constant traffic, including unicast, broadcast and multicast, circulating on the network at all times. This enables us to learn broadcast and multicast traffic and also detect successful handshakes as mentioned below. We achieve this by operating a node on the network which run scripts to send: traffic directly to the fuzzer's MAC address (e.g. raw data), multicast traffic (e.g. using mDNS), and broadcast traffic (e.g. ARP).
Verifying and resetting connections As the last message of the 4-Way Handshake is sent to the client, and hence our learner, the corresponding response will normally be a timeout, therefore we need to distinguish between the case where a handshake has finished successfully and other kinds of time-outs. As mentioned in the previous section, we operate a node on the network that constantly sends unicast data addressed to our learner's MAC address. Therefore, once a handshake is complete we observe these messages and can decrypt them to verify the contents. If this succeeds, we then check that the fuzzer can itself send encrypted data. This is done by sending an ARP-request for the MAC address of the gateway IP and waiting for an appropriate response.

Multi-core/Interface Sniffing and Injecting Due to the unreliability of Wi-Fi monitor mode for 802.11 frame injection and sniffing, we use two physically independent interfaces for each task—sending queries and sniffing for responses. We then have two processes running in parallel so that sniffing and injection can be carried out simultaneously. This is all implemented in Python using the Scapy[5] library.

5 Results

We used our adapted state machine learning algorithm to automatically learn 7 AP-side implementations of the 4-Way Handshake (see Table 2). In this section we will discuss the effect of our learning improvements, as well as the most notable results, including vulnerabilities, time behaviour and other interesting observations. This paper contains figures of two of the learned models, the rest are available online[6].

Time behaviour Three of the access points we tested exhibited the same timeout behaviour (3 retransmissions of message 1 and 3 with one second gaps, before ending with a deauthentication). Others had similar behaviour but over different times. One did not retransmit messages but timed out after 6 seconds (see Figure 3). Most interestingly, the Cisco WAP121 started sending encrypted data after 3 re-transmissions of message 3 over a period of 4 seconds. We discuss this in more detail in Section 5.2. We note that this finding in particular could not be detected by previous methods. The iOS model stands out in that it took significantly longer to learn than the others. The reason for this is that it

[5] http://www.secdev.org/projects/scapy/
[6] https://chrismcmstone.github.io/wifi-learner/

Model	Version	States	# Queries (Ex. error correction)	Error (%) Rate	Learn Time (hh:mm)
TP-Link WR841HP	V1_150519	6	963	5	1:32
Cisco WAP121	1.0.6.2	12	1163	4	1:42
TP-Link AC1200	140224	12	1113	8	2:35
iOS Personal Hotspot	8.1.3	6	887	2	5:46
ZxYEL AMG1302	V2	13	1684	1	1:53
D-Link DWRr600b	2.0.0EUb02	12	1113	1	1:18
Android hostapd	Oreo 8.0	12	1113	0.5	0:58

Table 2: Learning statistics for the Access Points we tested. Total queries excludes discards, total learning time includes time taken for error correction.

appears to silently timeout and hence hit the upper-bound timeouts mentioned in Section 4.2. Indeed, the implementation appears to be very minimalist, only responding to queries it considers to be correct, and ignoring those that are not. Nevertheless, this exemplifies the importance of the first stage of our learning method. By setting a small timeout (the 'normal response time'), when the learner carries out the equivalence checking stage, these queries will not suffer from this long timeout. Hence, thorough fuzzing was still possible, despite then having to relax this restriction for the second stage.

Non-determinism In Table 2 we state the error rate for each of the implementations we tested. We calculated this as the proportion of total executed queries that were detected as an error. An increased error rate had a direct effect on the time taken to learn. This is demonstrated by the TP-Link AC1200 which had an almost identical model to Android Hostapd, yet took over double the time to learn. In this particular case, the high error rate was due to the AP carrying over data from previous handshake executions with a relatively high probability.

Query reduction We were able to significantly reduce the number of queries required by the learning algorithm vs those actually posed to the SUL. Most of this reduction is down to the restrictions we enforce (i.e. delays after retransmissions (Section 4.2) and Wi-Fi specific optimisations (section 4.4). For example, the iOS model required over 20,000 queries in total but only 887 were actually queried, the rest predicted.

Similar models Our results reveal that three of the implementations appear to be very similar (TP-Link AC1200, Android Hostapd and D-Link DWRr600b). These models are somewhat different though, for example with respect to broadcast traffic, the DLink AP constantly broadcasted both Beacon frames and Probe Responses, whereas the TP-Link and hostapd only broadcasted Beacons. The implementations are also distinguished via their learning error rate, and as a result learning time. The TP-Link suffered from high error rate due to reasons stated above. Whereas, the other two APs had a very similar error rate.

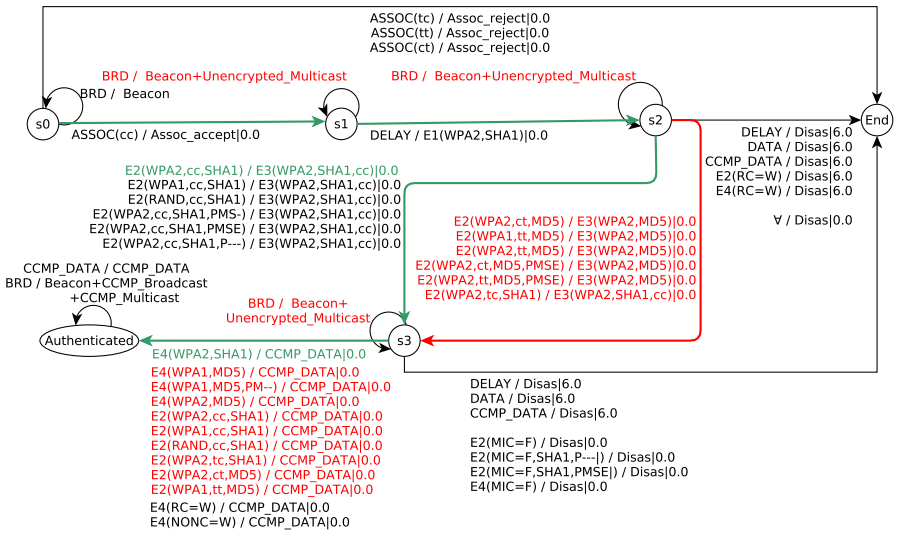


Fig. 3: State machine for the TP-Link TL-WR841HP, with normal transitions highlighted in green and those contributing to vulnerabilities in red.[7]

5.1 Encrypted Multicast Traffic Leakage

Using the broadcast/multicast learning feature of our framework, we discovered that that the TP-Link WR841HP transmits multicast data in plaintext when put in a certain state (see states 1 and 2 in Figure 3). More specifically, before a handshake is initiated, any multicast data will be sent encrypted with each unicast session key for all of the connected clients. However, during the execution of a 4-Way Handshake with a new client, and before the client has proven knowledge of the PSK (by sending a valid Message 2), this data will broadcast unencrypted to the client. Indeed, immediately after the 4-Way Handshake is completed, the data will only be sent encrypted. This represents a leakage of (potentially) sensitive multicast data.

5.2 Downgrade Vulnerabilities

In two access points we discovered downgrade attacks, namely for the Cisco WAP121 and the TP-Link TL-WR841HP.

Forcing Group Key encryption with RC4 Figure 3 shows the learned state machine implemented by the TP-Link WR841HP. We can see that despite initiating the connection in the Association stage with AES-CCM for both group and unicast keys, after starting the 4-Way Handshake using AES-CCM, the AP will surprisingly still accept a TKIP-formatted Message 2. In other words, if the

[7] In the interest of brevity we only include a selection of the most important transition labels. All queries that are 'disabled' are not included. We use the ∀ symbol to denote all input messages not specified in other transitions.

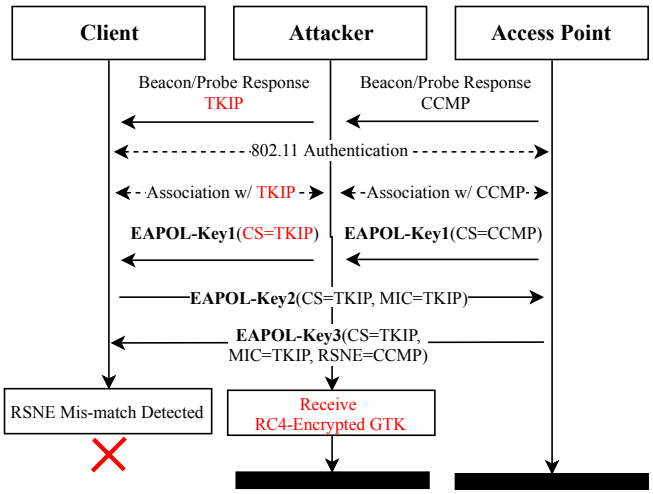


Fig. 4: Downgrade attack on the TP-Link WR841P to force encryption of the Group Key (GTK) with RC4

client switches ciphers mid-handshake, the AP will do also. This is indicated by the AP's response from state 3 to state 5, where it switches cipher suites to use TKIP's MD5 for the MIC, and encrypting the network's group key with RC4. Of particular significance is that this is in spite of the AP being set to exclusively use AES-CCM. Indeed, this is also advertised in the AP's Beacon and Probe Response messages.

To exploit this vulnerability, the adversary can set up their own AP with the same SSID as the target. This AP, however, only advertises support for TKIP in the beacons/probe response. As shown in Figure 4, the attacker will simultaneously carry out a 4-Way Handshake with the target AP, using AES-CCMP as the selected cipher. Messages will be selectively forwarded and altered between the target AP and client. Message 1 will contain the same nonce (for generation of the session key), but will be altered such that the cipher suite flag is set to TKIP. The client will generate its own nonce, calculate the session key, then send a TKIP MICed Message 2, which will be forwarded unchanged to the AP. This is accepted by the target and induces a downgrade to TKIP, resulting in a TKIP protected Message 3 response. The attacker will then observe this message, and can extract the RC4 protected GTK. The encrypted key could then be recovered and used for various attacks (see [30]).

AP-Side AES-CCMP to TKIP Downgrade Both the Cisco WAP121 and TP-Link WR841HP are vulnerable to AP-side downgrade attacks. That is, when both AES-CCMP and TKIP are supported by the AP and client, an attacker can force usage of TKIP. Normally, the client will always choose the more secure AES-CCMP.

With the Cisco WAP121, this vulnerability is indicated by the fact that Message 4, the message sent by the client to confirm the selected cipher, is not required by the AP. We can see from the state machine diagram Figure 5 in

Appendix A, after 3 re-transmissions of Message 3 over a period of 4 seconds, the AP will give up waiting for a response and complete the connection anyway. An illustrated exploitation of this vulnerability in depicted in Figure 6.

The affected TP-Link AP is also vulnerable to the same attack but in a slightly different way. That is, the AP does require a response to Message 3, but will accept a Message 2 in the place of Message 4. This enables an attacker to forge Message 4 by inducing a client to retransmit Message 2 and thereby carry out an AP-side downgrade attack.

For both APs, this attack is limited to downgrading the AP only. Correctly implemented clients will detect this downgrade through an inconsistency of the RSNE information which is selected in the Association stage and later encrypted and encapsulated within Message 3 from the AP. The client will decrypt the contents of the message, find that in fact the AP supports AES-CCMP and should then drop the connection.

Despite this, the flaw still represents a genuine vulnerability; any clients with existing connections could be forced to carry out a new 4-Way Handshake, e.g. due to roaming/signal loss or a client side deauthentication attack. Any data in the queue from the previous connection will then be secured with TKIP.

Disclosure TP-Link and Cisco have been fully informed of the vulnerabilities found, and in line with responsible disclosure, were given 6 months to address the vulnerabilities before publication. We also note that TP-Link no longer sell the vulnerable AP.

6 Conclusion

In this paper we introduced methods to handle the non-deterministic and timing related behaviour for lossy protocols such as Wi-Fi. These methods have been shown to be effective to infer models of numerous implementations of the 802.11 4-Way Handshake. This resulted in the discovery of several security vulnerabilities in widely used routers. The software will be made available as open source. In future work we want to extend the tool to handle the recently introduced WPA3. This uses the same 4-Way Handshake making it possible to use our tool on implementations of WPA3 with only minor changes.

We would like to apply our time learning technique to more protocols where time is important, particularly other protocols where long timeouts are present, making standard learning difficult to use. There are many security protocols where timing plays an important role, especially those running on unreliable mediums, such as other wireless protocols (Bluetooth, Zigbee, LTE), distance-bounding protocols (MasterCard's RRP, NXP's "proximity check"[8]), and others (DTLS, QUIC). We would also like to experiment with relaxing our assumptions. For instance, considering situations where multiple clocks are in operation.

Acknowledgements This work has been supported by the Netherlands Organisation for Scientific Research (NWO) through Veni project 639.021.750.

References

1. Aarts, F., de Ruiter, J., Poll, E.: Formal models of bank cards for free. In: Software Testing, Verification and Validation Workshops (ICSTW), Sixth International Conference on. IEEE (2013)
2. Aarts, F., Schmaltz, J., Vaandrager, F.: Inference and abstraction of the biometric passport. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. pp. 673–686. Springer (2010)
3. Aarts, F., Vaandrager, F.: Learning i/o automata. In: International Conference on Concurrency Theory. pp. 71–85. Springer (2010)
4. Angluin, D.: Learning regular sets from queries and counterexamples. Information and computation **75**(2), 87–106 (1987)
5. Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., Vigna, G.: SNOOZE: toward a stateful network protocol fuzzer. In: International Conference on Information Security. pp. 343–358. Springer (2006)
6. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A.: Model-based testing of reactive systems. In: Volume 3472 of Springer LNCS. Springer (2005)
7. Butti, L., Tinnes, J.: Discovering and exploiting 802.11 wireless driver vulnerabilities. Journal in Computer Virology **4**(1), 25–37 (2008)
8. Chothia, T., de Ruiter, J., Smyth, B.: Modeling and analysis of a hierarchy of distance bounding attacks. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/chothia
9. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE transactions on software engineering (3), 178–187 (1978)
10. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Combining model learning and model checking to analyze TCP implementations. In: International Conference on Computer Aided Verification. pp. 454–471. Springer (2016)
11. Fiterău-Broştean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F., Verleg, P.: Model learning and model checking of SSH implementations. In: 24th International SPIN Symposium on Model Checking of Software. SPIN 2017 (2017)
12. Fluhrer, S., Mantin, I., Shamir, A., et al.: Weaknesses in the key scheduling algorithm of RC4. In: Selected areas in cryptography. vol. 2259. Springer (2001)
13. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, pp. 379–395. Springer (2004)
14. Group, I..W., et al.: IEEE standard for information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications. IEEE Std **802**(11) (2010)
15. He, C., Mitchell, J.C.: Analysis of the 802.11 i 4-way handshake. In: Proceedings of the 3rd ACM workshop on Wireless security. pp. 43–50. ACM (2004)
16. He, C., Sundararajan, M., Datta, A., Derek, A., Mitchell, J.C.: A modular correctness proof of IEEE 802.11 i and TLS. In: Proceedings of the 12th ACM conference on Computer and communications security. pp. 2–15. ACM (2005)
17. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: International Conference on Computer Aided Verification. pp. 487–495. Springer (2015)
18. Janssen, M.: Combining learning with fuzzing for software deobfuscation (2016)

19. Jonsson, B., Vaandrager, F.: Learning mealy machines with timers http://www.sws.cs.ru.nl/publications/papers/fvaan/MMT/
20. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: Efficient test-based model generation for legacy reactive systems. In: High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International. pp. 95–100. IEEE (2004)
21. Mendonça, M., Neves, N.: Fuzzing Wi-Fi drivers to locate security vulnerabilities. In: 7th Dependable Computing Conference, EDCC. IEEE (2008)
22. Mitchell, C.: Security analysis and improvements for IEEE 802.11 i. In: 12th Annual Network and Distributed System Security Symposium (NDSS) (2005)
23. Niese, O.: An integrated approach to testing complex systems. Ph.D. thesis, Universität Dortmund (2003)
24. Raffelt, H., Steffen, B., Berg, T.: Learnlib: A library for automata learning and experimentation. In: Proceedings of the 10th international workshop on Formal methods for industrial critical systems. ACM (2005)
25. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: Learnlib: a framework for extrapolating behavioral models. International Journal on Software Tools for Technology Transfer (STTT) **11**(5), 393–407 (2009)
26. de Ruiter, J., Poll, E.: Protocol State Fuzzing of TLS Implementations. In: USENIX Security. vol. 15 (2015)
27. Shahbaz, M., Groz, R.: Inferring mealy machines. FM **9**, 207–222 (2009)
28. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017. pp. 276–287 (2017)
29. Tews, E., Beck, M.: Practical attacks against WEP and WPA. In: Proceedings of the second ACM conference on Wireless network security. pp. 79–86. ACM (2009)
30. Vanhoef, M., Piessens, F.: Predicting, decrypting, and abusing WPA2/802.11 group keys. In: USENIX Security Symposium (2016)
31. Vanhoef, M., Piessens, F.: Key reinstallation attacks: Forcing nonce reuse in WPA2. In: 24th ACM Conference on Computer and Communication Security (2017)
32. Vanhoef, M., Schepers, D., Piessens, F.: Discovering logical vulnerabilities in the Wi-Fi handshake using model-based testing. In: Asia Conference on Computer and Communications Security. ACM (2017)
33. Wang, L., Srinivasan, B.: Analysis and improvements over dos attacks against IEEE 802.11 i standard. In: 2nd Conference on Networks Security Wireless Communications and Trusted Computing (NSWCTC). IEEE (2010)

A Diagrams

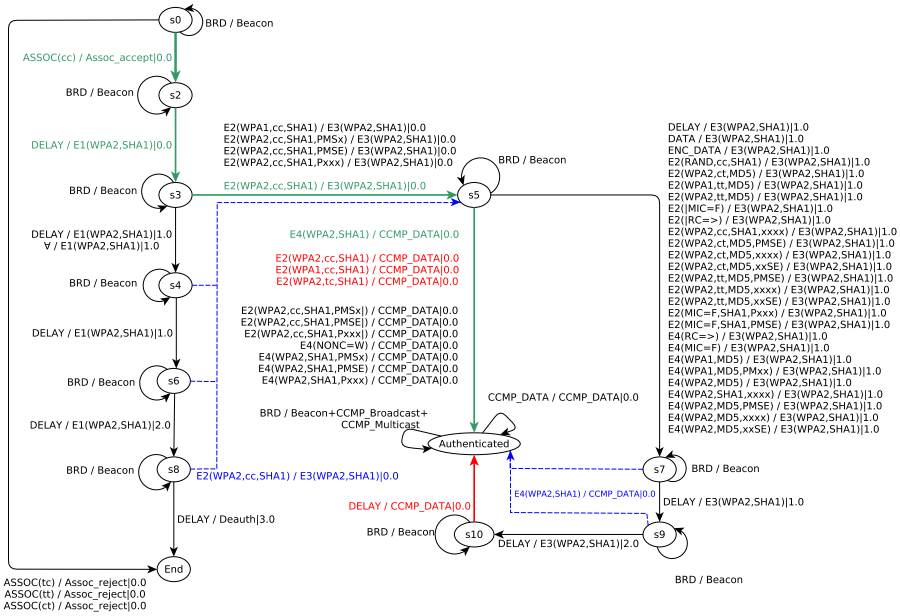


Fig. 5: Learned model for the Cisco WAP121. Note that for retransmission states, assumed transitions (as per Assumption 3) are represented by dotted blue lines.

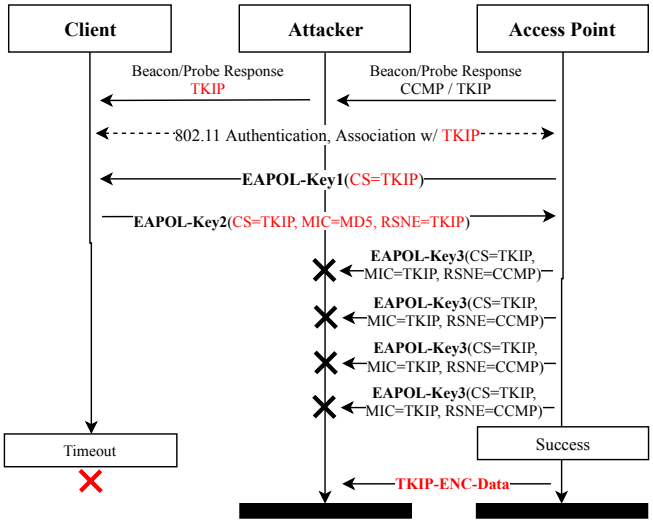


Fig. 6: Downgrade attack on the Cisco WAP121 to force usage of TKIP

B Calculations

We assume the probability of any error response is p_e, and that, for every query we have at least n responses. Therefore, the probability that i of the observed responses are correct is the the number of all possible combinations of i and $n-i$ responses times the probability of i correct responses and $n-i$ errors:

$$correct(i) = {}^{n}C_i p_e^i (1-p_e)^{n-i} = \tfrac{n!}{i!(n-i)!} p_e^{n-i} (1-p_e)^i$$

and the probability that the majority of observed responses are correct is: $mCorrect = \Sigma_{i=n/2\ldots n} correct(i)$. The probability that the correct output is the most commonly observed for m different queries strings is then $mCorrect^m$.

For the TP-Link AC1200, with an error rate of 8% and 1113 queries the chance of learning it correctly is 0.9925, for all other routers the probability of learning correctly was greater. Taking an average of 1000 queries and the average number of queries returned by our method (15), we see that a 10% error rate gives us a probability that the result is correct of 0.96, and with 100 tests and a 30% error rate the probability that they are all correct is 0.97.

Also important is the probability that our method will discard correct queries that has been correctly learned. We assume the worse case which is that there is only one incorrect message. In this case correctly learned queries are only discarded at the i^{th} test if, the at the $i-2^{th}$ test the incorrect response has been seen 1 time less than the correct message, and the incorrect response is seen for the next two messages.

It is only possible to have one less incorrect than correct message for an odd number of tests. The probably of this happening is at the 2m+1 th step:

$$oneOff(2m+1) = {}^{2m+1}C_m p_e^m (1-p_e)^{m+1} = \tfrac{(2m+1)!}{m!(m+1)!} p_e^m (1-p_e)^{m+1}$$

and the probably of correct queries being discarded at the 2m+1 th step as: $discard(2m+1) = oneOff(2m-1)p_e^2$. Following the discard of a correct state, there will be one more vote for the error response, therefore to return to the correct state and discard it again, we require 2 more correct responses than error respondences followed by 2 error responses to trigger the discard:

$$nextD(2m) = {}^{2m-2}C_{m-2} p_e^{m-2} (1-p_e)^m p_e p_e = \tfrac{(2m-2)!}{(m-2)!m!} p_e^m (1-p_e)^m$$

for $m \geq 2$.

The probability that the first discard of the correct query happens at a particular step is:

$$firstD(3) = (1-p_e)p_e p_e$$
$$firstD(2m+1) = discard(2m+1) - \Sigma_{i=1\ldots m-1}.firstD(2i+1).nextD(2(m-i)$$

So, therefore the probably of any discard of a correct response in the first n tests is:

$$AnyDiscard(x) = \Sigma_{i=1..x} firstD(x)$$