WiFuzz: detecting and exploiting logical flaws in the Wi-Fi cryptographic handshake

Mathy Vanhoef
imec-DistriNet[1]
mathy.vanhoef@cs.kuleuven.be

Encrypted Wi-Fi networks are increasingly popular. This is highlighted by new standards such as Hotspot 2.0 and Opportunistic Wireless Encryption. Hotspot 2.0 streamlines network discovery and selection, creating an authenticated roaming experience matching that of cellular phones. On the other hand, Opportunistic Wireless Encryption introduces unauthenticated encryption for Wi-Fi networks. However, these advancements are meaningless if there are implementation flaws in the 4-way Wi-Fi handshake that negotiates fresh session keys. In this report, we show how to detect and abuse logical flaws in implementations of this handshake.

Our goal is not to detect common programming errors such as buffer overflows or double frees, but to detect logical vulnerabilities. An example of a logical vulnerability is that some message(s) in a handshake can be skipped, causing it to use or negotiate an uninitialized (all-zero) cryptographic key. Clearly such vulnerabilities void all security guarantees. To detect these types of logical vulnerabilities, we first build a model of the Wi-Fi handshake that describes the expected behavior of an implementation. We then automatically generate invalid executions of the handshake, and check whether an implementation correctly reacts to these invalid executions.

We tested 12 Wi-Fi access points, and found irregularities in all of them. These consist of authentication bypasses, fingerprinting techniques, downgrade attacks, denial-of-service (DoS) attacks, and so on. Most prominently, we discovered two critical vulnerabilities in OpenBSD. The first can be abused as a DoS against the AP, and the second can be exploited to perform a man-in-the-middle attack against WPA1 and WPA2 clients. We also discovered downgrade attacks against MediaTek and Broadcom that force usage of TKIP and RC4. Additionally, we discovered a targeted DoS against Windows 7. We also found other irregularities in Airohive, Apple, Cisco, hostapd, and Windows 10.

[1]This report is primarily based on a paper co-authored with Domien Schepers and Frank Piessens [26], and on some novel research results. It is also based on paper co-authored with Frank Piessens [25]. All work was performed at the imec-DistriNet group of KU Leuven.

1 Introduction

More and more systems are using Wi-Fi to connect to the internet. Given the broadcast nature of these wireless transmissions, it is essential that this traffic is properly protected. The de facto method for this is Wi-Fi Protected Access 2 (WPA2) [1]. Interestingly, even upcoming standards such as Hotspot 2.0 and Opportunistic Wireless Encryption, rely on features of WPA2. Here, Hotspot 2.0 streamlines network discovery and selection, creating an authenticated roaming experience matching that of cellular phones. On the other hand, Opportunistic Wireless Encryption introduces unauthenticated encryption for Wi-Fi networks. All these encrypted networks rely on the Wi-Fi handshake to securely negotiate fresh pairwise keys. Once the handshake has completed, the pairwise keys are used to encrypt normal traffic. Therefore, a correct and secure implementation of the Wi-Fi handshake is essential to assure all transmitted data is properly protected. While there are some works that test Wi-Fi implementations for common programming mistakes such as memory overflows and use-after-frees, these only test the first stage of the handshake [18, 7]. Additionally, they are unable to detect logical vulnerabilities. In contrast, we show how to test various stages of the Wi-Fi handshake for the presence of logical implementation vulnerabilities.

To systematically test implementations, and detect logical vulnerabilities, we propose a model-based technique. First, we model the Wi-Fi handshake by defining the sequence of messages exchanged in a normal handshake (see Section 3). We then define a set of test generation rules that take this model (i.e., sequence of messages) and generate a set of invalid or unexpected handshake executions. For example, a test rule may generate an execution where a required message is skipped, or a message has an invalid integrity check. If an implementation does not reject such faulty executions, an irregularity has been detected. We then manually inspect all irregularities to determine whether they pose a security risk. By defining appropriate test generation rules, we can generate a representative set of test cases, which explore various edge cases in each stage of the Wi-Fi handshake.

We tested the 4-way handshake implementation of 12 Access Points (APs) from different vendors, and also performed a limited manual inspection of client implementations. This uncovered irregularities in all implementations, and revealed multiple vulnerabilities. The most serious vulnerability is a man-in-the-middle attack against the OpenBSD Wi-Fi client, which works against both WPA1 and WPA2. Other notable discoveries are downgrade attacks against Broadcom and MediaTek-based routers. Here, an adversary can force usage of WPA-TKIP instead of the more secure AES-CCMP. Also noteworthy are denial-of-service (DoS) attacks against Windows 7 and OpenBSD. Additionally, we found implementation bugs that, although they do not directly lead to practical attacks, are concerning. These results are surprising since the Wi-Fi handshake is not that complex, especially in comparison to other protocols such as TLS. In other words, even though the Wi-Fi handshake is fairly simple, our techniques still were able to detect a substantial number vulnerabilities.

2 Background

In this section we given a brief overview of existing fuzzing strategies, and give a brief history of the Wi-Fi Protected Access 2 (WPA2) standard.

2.1 Fuzzing Strategies

Researchers have proposed several techniques to detect vulnerabilities in programs. One prominent technique is fuzzing, where a program is fed random input, and then monitored to see whether it crashes or hangs. In variants of this approach, grammars are used to generate properly formatted inputs. This enables encoding of application-specific knowledge and test heuristics. Another variation is used by the AFL fuzzer, where promising inputs are detected based on which (previously unreached) code blocks they explore [27]. While these techniques can be effective and easy to use, they have several limitations. For example, the conditional statement "`if (x==137) then`" only has one in 2^{32} chance of being fulfilled if x is a random 32-bit input value. Hence serious vulnerabilities may go undiscovered. Symbolic execution overcomes this problem by running the program not on random concrete inputs, but on symbolic inputs. Now when a conditional statement is encountered, analysis is essentially forked over all feasible branches. Every fork has a path constraint which records the conditions that the input must satisfy for an execution to reach this location in the program. To determine whether a branch is feasible, and to generate concrete inputs that follow certain branches, a SAT or SMT solver is used. Several tools implement this technique, examples are DART [13], KLEE [8], and SAGE [14]. Symbolic execution has two main limitations. First, certain queries to the SMT solver can be slow to practically unsolvable. The second problem is a state explosion when analysing large programs, where there are an exponential number of execution paths in the program.

While the previous techniques can find common program mistakes, such as buffer overflows and double frees, they cannot detect flaws in the application logic of a program. To detect those, we need a model that defines the expected behaviour of a program. For example, Beurdouche et al. first constructed a model of TLS by specifying the order in which packets should be transmitted, i.e., they first defined the state machine of TLS [3]. Then they tested implementations to see whether their behaviour deviated from this model. De Ruiter and Poll used a different technique to uncover logical bugs [11]. They first defined the set of valid TLS packets that can be sent, and then used this to reconstruct the state machine of an implementation. This state machine then had to be manually inspected for vulnerabilities.

2.2 History of Wi-Fi Protected Access 2 (WPA2)

Initially, the 802.11 standard provided Wired Equivalent Privacy (WEP) to protect data. Unfortunately, it contained major design flaws, and is considered completely broken [12, 21, 4]. To address these security issues, the IEEE designed both a short-term and long-term solution. Their long-term solution is called (AES-)CCMP. It uses AES in counter mode for encryption, and CBC-MAC for authenticity. Unfortunately, older WEP-compatible devices were not performant enough to implement CCMP in software using firmware updates. To solve this problem, the (WPA-)TKIP protocol was designed as a short-term solution. Similar to WEP, it is based on the RC4 cipher, meaning WEP-capable hardware could support it using only firmware upgrades.

Due to the slow standardization process of the IEEE, the Wi-Fi Alliance already began certifying devices based on a draft version of the 802.11i amendment. This certification program was called Wi-Fi Protected Access (WPA), and required support for TKIP, but did not mandate support for CCMP. In this paper we will refer to it as WPA1. Unfortunately, this led to the common misunderstanding that WPA1 is synonymous with using TKIP. Once 802.11i was standardized, the Wi-Fi Alliance started the WPA2 certification program. This certification requires that a device supports CCMP, but does not mandate support for TKIP. This led to another common misconception, namely that WPA2 means (AES-)CCMP is used, while in reality a WPA2 network might still use (or support) TKIP.

Since WPA1 and WPA2 are both based on the 802.11i standard, they are nearly identical to each other. Therefore, unless mentioned otherwise, we will treat WPA1 as identical to WPA2. The few differences between them are discussed in Section 3 and in our paper [26]. Finally, we use the term Robust Security Network (RSN) to refer to 802.11i security mechanisms in general (i.e. it can refer to both TKIP and CCMP).

3 The Wi-Fi Handshake

In this section we present our model of the Wi-Fi handshake, which will be uses as the basis for our testing technique [26].

3.1 Stage 1: Network Discovery

An Access Point (AP) periodically broadcasts beacons to advertise its presence. This is illustrated in stage 1 of Figure 1. These beacons include the supported link-layer encryption algorithms (i.e., the supported ciphers) that are supported by the AP. Generally this is either TKIP and/or CCMP.

Clients can discover networks by passively listening for beacons, or by actively sending probe requests. This is also illustrated is stage 1 of Figure 1. Both beacons and probe responses contain the name and capabilities of the wireless network. In particular this includes the Robust Security Network information Element (RSNE), which defines the supported pairwise cipher suites of the network, and the group cipher suite that is used. Note that the cipher suite can be either TKIP or CCMP. Although the bit-wise encoding of the RSNE differs between WPA1 and WPA2, in both WPA versions the RSNE contains the same information.

3.2 Stage 2: Authentication and Association

Once the client has found a network to connect to, the actual handshake starts (see stage 2 of Figure 1). Here the client is called the supplicant, and the access point is called the authenticator (we treat these terms as synonyms). In principle, authentication may already happen at this point, but in practice most networks use Open System authentication. This mechanism allows any client to authenticate. In a WPA or WPA2 network, actual authentication will be performed at stage 4 during the 4-way handshake.

Once “authenticated”, the supplicant sends an association request to the AP. This frame includes the pairwise cipher that the client wants to use, encoded in an RSNE element. If the supplicant encodes the RSNE using the conventions of WPA1, the WPA1 variant of the handshake will be executed. Otherwise, the WPA2 variant will be executed. Because the AP also uses an RSNE element to advertise its *supported* list of cipher suites, we will use the term RSNE-Chosen when the RSNE encodes the *chosen* cipher suites. The AP replies with an association response, indicating the association was successful or not.

3.3 Stage 3: 802.1x Authentication

The third stage is optional, and consist of 802.1x authentication to a back-end Authentication Server. For example, this may consist of authentication using a username and password to a RADIUS server. The end result of this authentication is that the client and AP share a secret Pairwise Master Key (PMK). Since parts of 802.1x have already been tested in other works [2, 16, 9, 19, 5], or are based on TLS [6, 3, 11, 20], we do not test this further in our work. Instead, we simply assume the supplicant and authenticator derive the PMK from a secret pre-shared key.

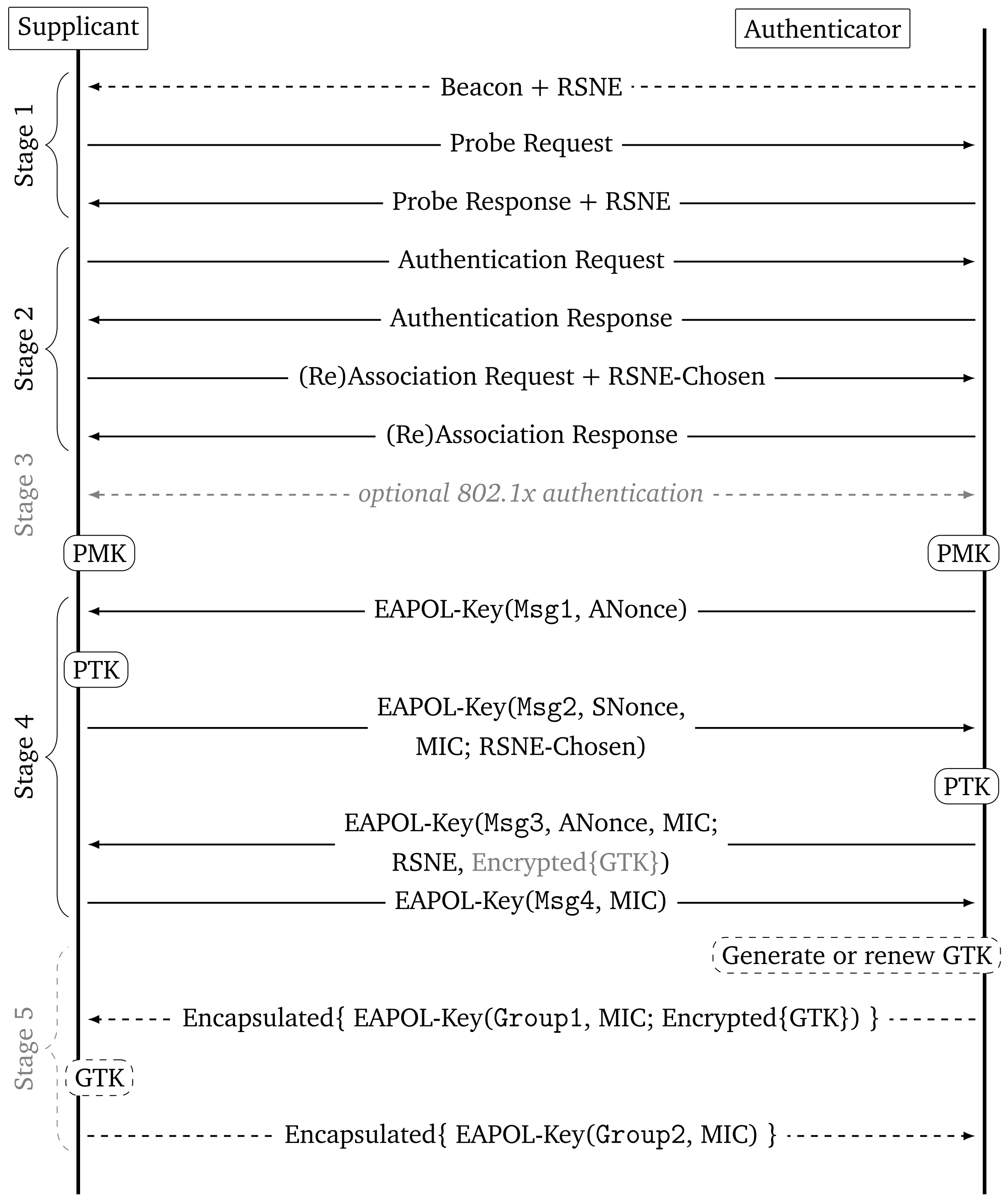

Figure 1: Frames sent in the 802.11 handshake, including their most important parameters [26]. Optional stages or parameters are shown in (dashed) gray.

Protocol Version 1 byte	Packet Type 1 byte	Body Length 2 bytes
Descriptor Type – 1 byte		
Key Information 2 bytes	Key Length 2 bytes	
Key Replay Counter – 8 bytes		
Key Nonce – 32 bytes		
EAPOL Key IV – 16 bytes		
Key RSC – 8 bytes		
Reserved – 8 bytes		
Key MIC – variable		
Key Data Length 2 bytes	Key Data variable	

Figure 2: Layout of EAPOL-Key frames [10, §11.6.2].

3.4 Stage 4: The 4-way Handshake

The fourth stage consists of the 4-way handshake, which provides mutual authentication and negotiates a fresh Pairwise Transient Key (PTK). Additionally, it prevents downgrade attacks by cryptographically verifying the RSNEs received during the network discovery and association stage. The PTK is derived from the Authenticator Nonce (ANonce), Supplicant Nonce (SNonce), and the MAC addresses of the client and AP.

Messages in the 4-way handshake are defined using EAPOL-Key frames (see Figure 2). We will briefly discuss the most important fields. First, the descriptor type determines the remaining structure of an EAPOL-Key frame. Although WPA1 uses the value 254 for this field, and WPA2 uses the value 2, both define an identical remainder structure of the EPAOL-Key frame. Following this 5-byte header is the key information field. It consists of a 3-bits key descriptor version subfield, and eight (one-bit) flags called the key info flags. The key descriptor field defines the cipher suite that is used to protect the frame. This is either AES with HMAC-SHA1, or RC4 with HMAC-MD5. If the client selects CCMP as the pairwise cipher, this must be AES with SHA1. Otherwise, if the client selects TKIP, RC4 with MD5 must be used. The replay counter field is used to detect replayed frames. When the client replies to an EAPOL-Key frame of the AP, it must use the same replay counter as the one in the previously received EAPOL-Key frame from the AP. The AP always increments the replay counter after transmitting a frame. Finally, the integrity of an EAPOL frame is protected using a Message Integrity Check (MIC), and the key data field is encrypted if it contains sensitive data. Recall that the encryption algorithm used to encrypt the key data field is specified in the key descriptor field.

When using WPA2, the receiver of an EAPOL-Key frame can distinguish among the different messages of the 4-way handshake by inspecting the key info flags. We use the notation message n to refer to the n-th message of the 4-way handshake. Note that in Figure 1 we use the following notation:

EAPOL-Key(`MsgType`, Nonce, MIC; Key Data)

This represents a frame of type `MsgType`, with the given nonce (if present). When the MIC parameter is present, the frame is authenticated using a message integrity check. Finally, all parameters after the semicolon are stored in the key data field (see Figure 2). The notation Encrypted{GTK} is used to stress that, if the GTK is included, it must be encrypted using the PTK.

3.4.1 Message 1

The first message of the 4-way handshake stage is sent by the AP, and contains the randomly generated ANonce of the AP. The key info flags that must be set in this message are `Pairwise` and `Ack`, which are represented by the label `Msg1`. This message is not protected by a MIC, and hence can be forged by an attacker. When the client receives this message and learns the ANonce, it posses all information to calculate the PTK.

3.4.2 Message 2

This message contains the random SNonce of the supplicant, and is protected using a MIC. The key info flags that must be set are `Pairwise` and `MIC`, which are represented by the label `Msg2`. The Key Data field includes the authenticated RSNE-Chosen element, which contains the chosen cipher suites previously sent in the (re)association request.

When the AP receives this message, it calculates the PTK, verifies the MIC, and compares the (authenticated) RSNE-Chosen with the one previously received in the association request. If they differ, a downgrade attack has been detected, and the handshake is aborted.

3.4.3 Message 3

Message 3 is send by the AP, and its required key info flags are `Pairwise`, `MIC`, and `Secure`. Here the Key Data field includes the RSNE, which contains the supported cipher suites of the AP. Additionally, if WPA2 is used, it also includes the encrypted GTK. When WPA1 is used, the GTK is sent to the supplicant using a group key handshake (see Section 3.5). When the client receives this message, it checks that the (authenticated) RSNE in message 3 is identical to the one received in beacons and probe requests. If they differ, a downgrade attack was attempted, and the handshake is aborted.

3.4.4 Message 4

The supplicant sends message 4 to the authenticator, to confirm that the handshake has been successfully completed. This last message is also authenticated using a MIC. When WPA2 is used, the required key info flags are `Pairwise`, `MIC`, and `Secure`. However, for WPA1, the required key flags does not include `Secure`. We use `Msg4` to represent the required key info flags for both WPA versions. Note that message 2 and message 4 have the same required key info flags if WPA1 is used. The only way to differentiate message 2 and 4 in WPA1 is to see whether there is data present in the key data field. Once the authenticator received message 4, normal (encrypted) data frames can be transmitted.

3.5 Stage 5: Group Key Handshake

The last stage consists of the group key handshake, and is required when using WPA1. It transports the group key to the client, which is used to protect broadcast and multicast traffic. In both WPA1 and WPA2, the group handshake is also periodically executed to renew the group key. Note that in Figure 1 we use the notation Encapsulated$\{\cdot\}$ to denote that the complete EAPOL-Key frame is also protected using either TKIP or CCMP.

4 Our Model-Based Testing Technique

In this section we present our model-based testing technique. We start with a high-level description, discuss the features and properties of the handshake that will be tested, and finally explain how the tests are executed in practice [26].

4.1 General Approach

Our goal is to perform black-box tests of implementations, and detect logical vulnerabilities in all stages of the handshake. We accomplish this by taking the Wi-Fi handshake as described in the previous section, and applying several (invalid) modifications to it. Each modification that can be applied is described by a test generation rule, and results in a test case that can be executed in a black-box manner.

Every test case is defined by the sequence of messages that must be transmitted to the AP, and the expected replies. Additionally, it defines whether this exchange of messages should result in a successful connection or not. For example, skipping a message should result in a failed connection, whereas retransmitting a message should still result in a successful connection. To construct a concrete test case, we start from a normal handshake (i.e., the model) as defined in Section 3. Such a normal execution of the handshake can already be treated as (trivial) test which should always succeed. Note that a normal handshake can use either WPA1 or WPA2, and negotiates either TKIP or CCMP as the pairwise cipher. Additional tests are generated by changing such a normal handshake execution according to a given test generation rule. These rules modify the execution of the handshake, with as goal to determine how the AP reacts to various (in)valid modifications of the handshake.

When a generated test case has failed, manual inspection is required to determine the precise type of bug present in the implementation. Here additional black-box test can be performed, or the source code can be inspected if it is available. Once the cause of the bug is identified, we determine whether an adversary can abuse it in an actual attack.

Defining appropriate test generation rules is one of the most critical steps in our testing method. After all, they determine the types of bugs and vulnerabilities that can be discovered. In this report, our test generation rules are inspired by an analysis of the Wi-Fi specification, rudimentary code inspections, and already known (and patched) implementation vulnerabilities. Apart from these automated tests, we also briefly reviewed the source code of implementations that support both a client and an AP mode.

4.2 Test Generation Rules

Our first category of test generation rules manipulate messages as a whole. In particular, we define two test generation rules in this category:

1. **Dropped messages:** Generate a set of test cases where each message, together with its expected responses (if any), is removed.

2. **Injected messages:** Generate a set of test cases where each message allowed in a handshake, together with its expected responses (if any), is inserted before every message that is normally transmitted.

Our second set of test generation rules change (implicit) parameters of handshake messages. In particular, we came up with the following set of rules:

3. **Invalid RSNE (cipher suite):** Generate test cases where the association request and message 2 have a modified RSNE. This test covers all possible cipher suite combinations, where valid values for the pairwise cipher are TKIP, CCMP, or TKIP/CCMP. The group cipher suite is either TKIP or CCMP.
4. **Invalid EAPOL-Key descriptor type:** Generate test cases where the descriptor type in each EAPOL-Key frame is switched to an unexpected value. In particular, we switch value 2 (used by WPA2) with value 254 (used by WPA1), and visa versa. We also generate test cases with a random value other than 2 and 254.
5. **Invalid EAPOL-Key key info flags:** Generate test cases that together try all possible combinations of key info flags. Recall that the eight key flags are: `Pairwise`, `Install`, `Ack`, `MIC`, `Secure`, `Error`, `Requested` and `Encrypted`. This results in a total of 2^8 possible combinations for every EAPOL-Key frame in the handshake.
6. **Switched EAPOL-Key cipher suite:** Generate test cases where each EAPOL-Key message is protected using an unexpected cipher suite. That is, AES with SHA1 is replaced by RC4 with MD5, and visa versa (recall Section 3.4).
7. **Invalid EAPOL-Key replay counter:** Generate test cases where the replay counters in EAPOL-Key frames are either lower or higher than the expected replay counter.
8. **Invalid EAPOL-Key nonce:** Generate test cases where a nonce is added to EAPOL-Key frames that should not be containing a nonce.
9. **Invalid EAPOL-Key MIC:** Create test cases where each EAPOL-Key frame has an invalid MIC, with the `MIC` flag either set or not set in the key info flags.

To prevent an exponential explosion of the number of generated tests, we do not combine different test generation rules. However, inspired by a DoS attack that poisons the supplicant with a forged ANonce [15], we do combine rule 2 and 8 to obtain the following test generation rule:

10. **Injected Nonce**: Inject a forged message 1 or message 2 after, and before, a valid message 1 or message 2, respectively. This message contains a random nonce and an invalid MIC, with the `MIC` flag either set or not set in the key info flags.

Applying the above test generation rules always results in a finite number of test cases. Apart from rule 8, these are all deterministic test generation rules. This has the advantage that if we can rerun all the test cases, we will obtain the same results. Hence our technique, in contrast with random fuzzing, creates repeatable results.

4.3 Executing Test Cases

We implemented a test harness that executes test cases against Access Points (APs). It transforms each abstract message of a test case into a concrete Wi-Fi frame. For example, it fills in MAC addresses and nonce values, and encrypts the appropriate fields of the frame using the current pairwise key. To accomplish this, some state information needs to be maintained. For instance, upon receipt of message 1, the generated PTK is stored.

To confirm that the AP is still working before executing a text case, we first wait for a beacon. Additionally, if a deauthentication message is received during the execution of a test case, we can already conclude the handshake has failed. Finally, timeouts are used to detect when the AP ignored a frame.

4.4 Validating and Resetting the Connection

After running a test case, we have to determine whether the handshake resulted in a valid connection or not. This can then be compared to the expected result. For example, some test generation rules should not negatively impact a handshake execution (e.g. retransmitting messages), while other rules should result in a failed connection (e.g. dropping messages). Merely listening for deauthentication frames, which are transmitted when the authenticator aborts an ongoing handshake, is not sufficient. This is because it may be that, according to the AP, the handshake is not yet completed, meaning it is still waiting for certain messages.

To verify if a connection was established or not, we send an ARP request to the gateway, where we request the MAC address of the gateway. Note that the gateway's IP address is known by the test harness. If the connection was successful, the gateway will reply to the sender MAC address of the ARP request. Finally, after executing a test case, we reset any state at the AP by sending a deauthentication message. This assures that different test cases do no influence each other.

5 Discovered Attacks

We now present the results of our testing technique, which we used to test 12 different APs (see Table 1). First we discuss general vulnerabilities that were discovered in several devices, and then we list device-specific vulnerabilities. Note that in this report the focus is on exploitable vulnerabilities, other findings can be found in [26]. Proof-of-concepts of selected attacks are available online [23].

Table 1: Implementations that were tested for logical vulnerabilities.

Impl. Name	Version	Hardware
Broadcom	5.10.56.46	RT-N10
Broadcom	5.10.85.0	WAG320N
Hostapd	2.6	TL-WN722N
OpenBSD	6.0 generic 2148	WL-172
Telenet	Ver 30.10.2016	Home gateway
MediaTek	3.0.0.9	RT-AC51U
Windows 7	build 7601	TL-WN722N
Windows 10	build 10240	TL-WN722N
Apple Airport	7.6.7	Time Capsule
Apple macOS	10.12 (Sierra)	MacBook Pro
Aerohive	Ver 1.11.2016	HiveAP 330
Aironet (Cisco)	Ver 1.11.2016	Aironet 1130 AG

5.1 Fingerprinting Mechanisms

Any irregularities in how an AP sends or handles messages can be used to fingerprint and hence identity the implementation being used. Here, we discovered two fields of an EAPOL-Key frame that are particularly useful for this purpose: the key info field, and the descriptor type field. First, most vendors require, or prohibit, different key info flags in messages received during the Wi-Fi handshake. This unique combination can form a fingerprint of an implementation. Second, the descriptor type field is supposed to contain the value 2 if WPA2 is used, and 254 if WPA1 is used. Recall from Section 3 that in practice these values are equivalent. Nevertheless, not all implementations treat these values as being identical. Some require that the value matches the type of handshake being executed, some allow both values, and even others allow *any* byte value. This can be used to identify a specific implementation.

In order to fingerprint APs based on the key info field or descriptor type field, we must be able to execute a (partial) Wi-Fi handshake with the AP. This is only possible when possessing the necessary credentials to the network. We do not consider this a significant limitation. For example, in an enterprise network such as eduroam, devices may have access to the network, but they are not considered trusted. In other words, there are indeed situations where an *adversary* can execute a Wi-Fi handshake with a targeted AP.

5.2 Impossible TKIP Countermeasures DoS

Several APs incorrectly implement the TKIP countermeasure procedure. This procedure is a defense mechanism designed to mitigate weaknesses in TKIP. Simplified, it is activated when the AP receives two MIC failure reports, after which no client can connect to the network for one minute. We discovered several flaws in how MIC failure reports are handled by implementations. Note that an attacker can cause a client to send a failure report by capturing a TKIP packet, modifying it, and then injecting the packet [24]. Injecting MIC failure reports in a CCMP-only network requires the necessary credentials to connect to the network.

When a client is not using TKIP, it should not be sending MIC failure reports to the AP. Nevertheless, Broadcom, Windows 10, and Aerohive APs, accept MIC failure reports even if the client is only using CCMP. This activates the TKIP countermeasures, where the AP disallows new connections for one minute. Hence, an adversary can abuse this as a denial-of-service attack against a CCMP-only network. We also found that Windows 10 and OpenBSD permanently block connections once the countermeasure procedure is started. This results in a permanent DoS attack. Only after restarting the Windows 10 or OpenBSD AP are connections allowed again.

Strangely, Broadcom's implementation accepts MIC failure reports, and initiates a new TKIP countermeasure period, *even when an existing TKIP countermeasure period is already in progress*. When abusing this to trigger multiple simultaneous countermeasure periods, the AP eventually crashes and becomes permanently unresponsive.

5.3 OpenBSD

OpenBSD contains several vulnerabilities in their implementation of the Wi-Fi handshake, and in particular in the 4-way handshake. These lead to two practical attacks, namely a denial-of-service and a man-in-the-middle attack.

5.3.1 Permanent TKIP countermeasure DoS against the AP

OpenBSD contains two vulnerabilities that, when combined, create an effective DoS attack against the AP. These two vulnerabilities are:

1. OpenBSD does not terminate the TKIP countermeasures after one minute, meaning the AP will permanently be unusable after it receives two MIC failure reports.
2. OpenBSD accepts TKIP MIC failure reports at any time during the handshake, even before the client is authenticated, and session keys have been negotiated. On reception of a MIC failure report sent before completion of the 4-way handshake, the AP will use an all-zero session key to verify the authenticity of the MIC failure report.

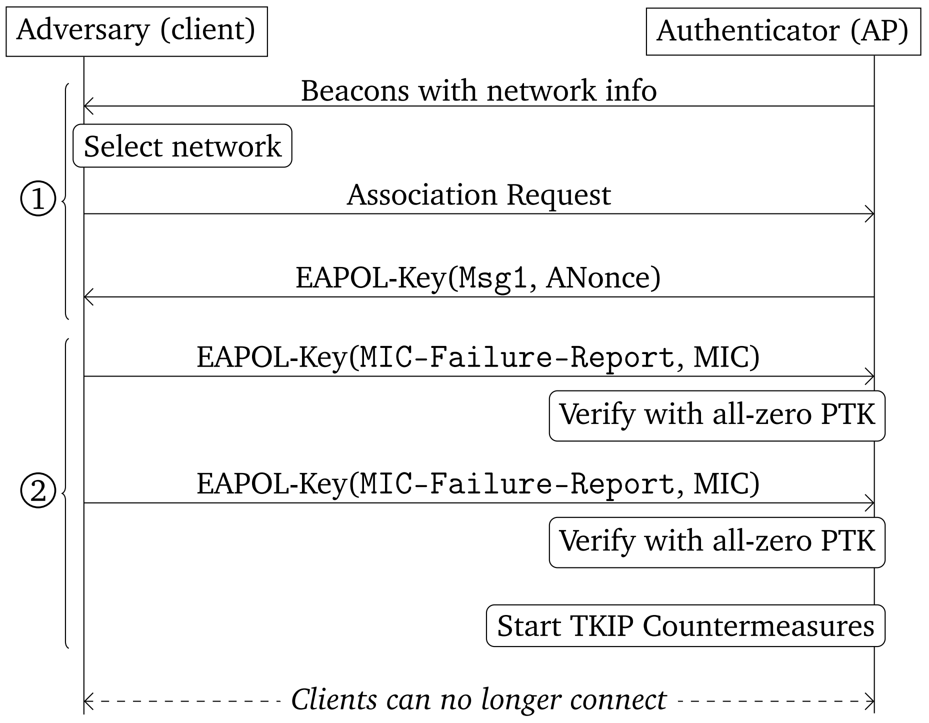
Adversary (client)
Authenticator (AP)
Beacons with network info
Select network
①
Association Request
EAPOL-Key(Msg1, ANonce)
EAPOL-Key(MIC-Failure-Report, MIC)
Verify with all-zero PTK
②
EAPOL-Key(MIC-Failure-Report, MIC)
Verify with all-zero PTK
Start TKIP Countermeasures
Clients can no longer connect

Figure 3: Denial-of-Service attack against an OpenBSD AP. Note that MIC failure reports are EAPOL-Key frames that have the `MIC`, `Error`, and `Request` key info flags set.

This can be exploited by sending two MIC failure reports to the AP, after receiving message 1 of the 4-way handshake. These two MIC failure reports are authenticated using an all-zero session key. The OpenBSD AP will accept these frames, and will start the TKIP countermeasures. As a result, the AP becomes permanently unusable. Hence an adversary can easily take down an OpenBSD AP, where connectivity can only be restored by restarting the AP. Note that an adversary does not require any credentials to execute this attack. Figure 3 illustrates the attack. The first stage corresponds a normal execution of the handshake, but at stage two the adversary (unexpectedly) sends two MIC failure reports.

5.3.2 Man-in-the-Middle Attack

Surprisingly, manually inspecting the OpenBSD source code revealed that its Wi-Fi client implementation does not contain a state machine. Put differently, it's not checked whether handshake messages (and EAPOL-Key messages in general) are being received in their proper (normal) order. This means that if an adversary would send an unexpected message to the client, OpenBSD will try to process it, instead of ignoring it.

Figure 4 shows how an adversary can abuse this vulnerability by setting up a rogue AP, and performing a man-in-the-middle attack against a victim. In the first stage of the

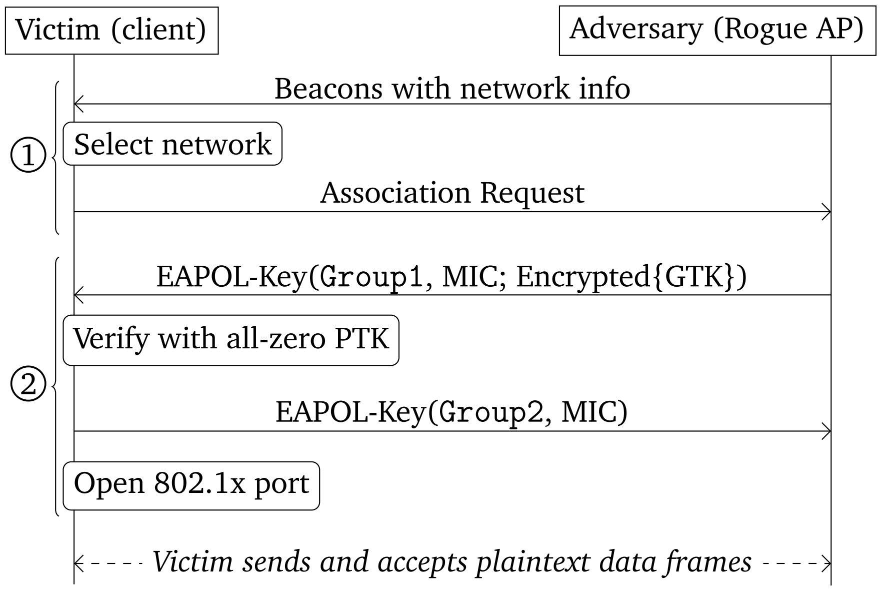

Figure 4: Man-in-the-middle attack against OpenBSD by setting up a rogue AP.

attack, the Wi-Fi handshake is executed normally. However, in stage 2 of the attack, instead of sending the first message of the 4-way handshake, the adversary sends a Group Message 1. Due to the missing state machine of the OpenBSD client, it will accept and process the Group Message 1. Here, the client will try to verify the message integrity check (MIC) of this frame using an all-zero session key. Naturally, the adversary can easily forge this MIC. As a result, the client replies using Group Message 2, and will then open the 802.1x port. Opening this port means the client starts accepting and transmitting data frames. Because no session key (PTK) was installed before opening this port, the client will transmit (and accept) plaintext frames. Note that the rogue AP was never authenticated, since we skipped the 4-way handshake.

5.4 Broadcom: Downgrade Attack by Forging Message 4

We discovered that Broadcom cannot distinguish message 2 and message 4 of the 4-way handshake when using WPA1. This is because, in the WPA1 variant of the 4-way handshake, both these messages have the same key info flags that must be set (recall Section 3). The only difference between them is that message 2 includes data in the key data field of the EAPOL-Key frame, while message 4 does not. However, when using WPA1, Broadcom does not check for this difference.

5.4.1 Downgrade Attack

We can abuse this flaw to trick the AP into using TKIP as follows. First, the adversary sets up a rogue AP that acts as a man-in-the-middle between the client and AP (see

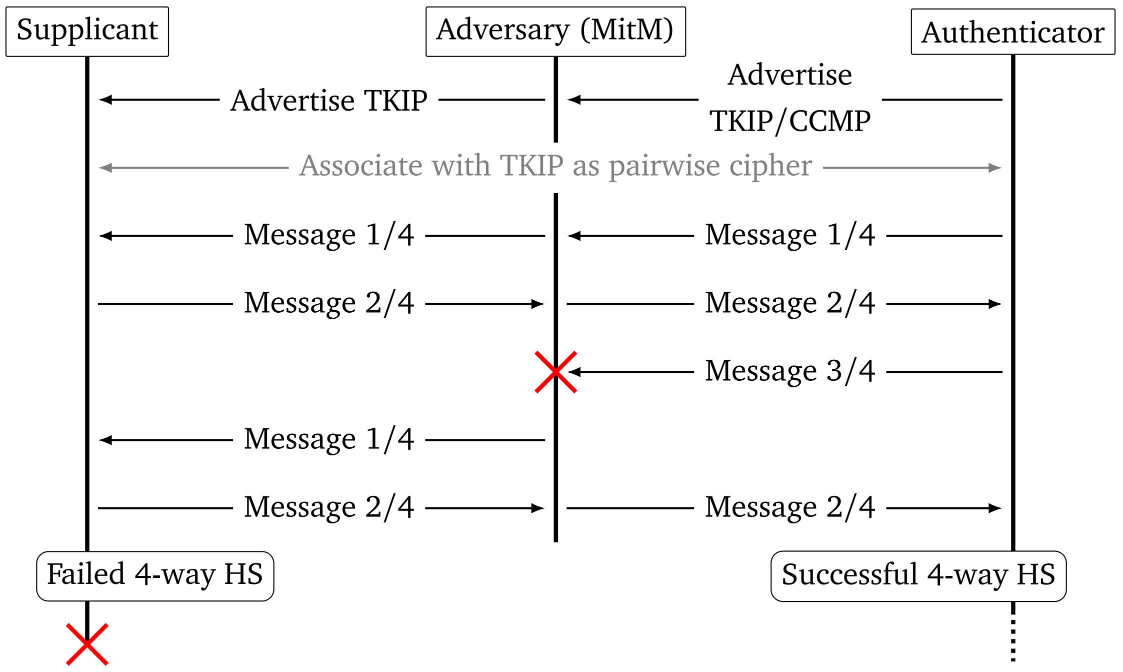

Figure 5: Downgrading a Broadcom authenticator to TKIP, when both TKIP and CCMP is enabled. The attack only works against WPA1, since Broadcom only confuses message 2 and 4 in WPA1.

Figure 5). She modifies all beacons and probe responses, so it appears that the network only supports TKIP. As a result, the client will connect to the AP and request TKIP as the pairwise cipher. At this point the adversary will forward message 1 and 2 of the 4-way handshake without modification. However, it will block message 3, assuring that the supplicant never sees this message. Blocking this message is essential since it contains the real RSNE (supported cipher list) of the AP, which includes both TKIP and CCMP. This RSNE differs from the one that the adversary advertised in beacon and probe requests. The client would abort the handshake if this difference is detected.

The adversary now induces the client into retransmitting a valid message 2, by forging an unauthenticated message 1. When the client receives the forged message 1, it transmits a new message 2. The retransmitted message 2 is forwarded to the AP, which will be wrongly treated as being a (valid) message 4. The AP now thinks the 4-way handshake has been successful, and installs the session keys to enable transmission of normal (encrypted) traffic. In particular, it will transmit the first message of the group key handshake, and encrypt it using TKIP. Note that the client will ignore this group key message, because it never received message 3 of the 4-way handshake. Nevertheless, it is problematic that the AP is using TKIP.

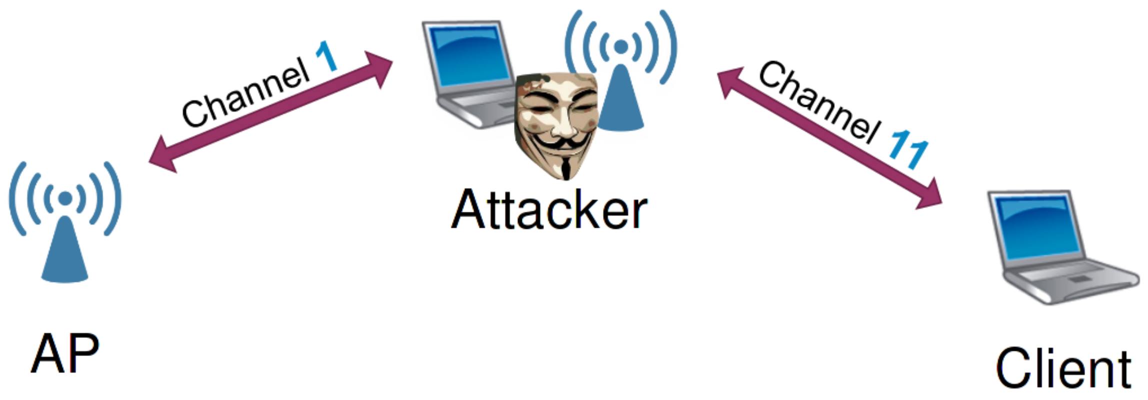

Figure 6: Graphical illustration of a channel-based MitM attack.

5.4.2 Discussion

Strangely, based on the 802.11 standard, the ability to forge message 4 should not introduce practical attacks. More precisely, the 802.11 standard states that message 4 is only required for reliability and not security [17, §11.6.6.8]:

> "While Message 4 serves no cryptographic purpose, it serves as an acknowledgment to Message 3. It is required to ensure reliability and to inform the authenticator that the supplicant has installed the PTK [..]"

However, our attack against Broadcom shows that message 4 is essential in preventing downgrade attacks against the 4-way handshake.

In order to execute the attack in practice, the adversary must obtain a MitM position between the client and AP. This is not possible using a rogue AP with a MAC address different from the real AP, because the negotiated session keys are tied to the MAC addresses of the client and AP. Our solution is to use a channel-based MitM attack, where the adversary clones the AP on a different channel [25]. Figure 6 illustrates such a channel-based MitM attack. With this approach, we are sure clients will never directly communicate with the real AP. Additionally, bot the client and AP will generate the same session key, meaning message 2 will have a valid MIC when processed by the real AP.

5.5 MediaTek

We observed that MediaTek does not verify the RSNE in both message 2 and 3 of the 4-way handshake. Therefore, a MediaTek client can be downgraded into using WPA-TKIP instead of AES-CCMP. This enables an adversary to exploit weaknesses in WPA-TKIP, examples attacks being those in [22] and [24]. We consider this especially problematic because MediaTek's client functionality is used to extend the range of another AP. Consequently, during an attack the traffic of all devices connected to the MediaTek router will be forwarded over a connection that has been downgraded to TKIP.

MediaTek is also affected by a DoS attack that poisons the AP with an invalid SNonce. In particular, an adversary can inject a message 2 with a random nonce and invalid MIC, right after the client transmitted the real message 2. In response to this message, the MediaTek AP will generate, *and store*, a new session key derived from the forged SNonce. This causes the handshake between the real client and AP to time out and fail.

5.6 Windows

Against Windows 7 we discovered an unauthenticated targeted DoS attack that permanently prevents a *specific* client from connecting to the network. To execute the attack, an adversary has to send two association requests right after one another, with as sending MAC address the targeted victim. After this, the victim can no longer connect to the network. In contrast to our other DoS attack, this one allows an adversary to block *specific* MAC addresses from connecting to the network.

Finally, recall from Section 5.2 that Windows 10 is vulnerable to our impossible TKIP countermeasure attack. Against Windows 10, this even results in a permanent DoS.

5.7 Aerohive

As mentioned in Section 5.2, we found that a client can trigger the TKIP countermeasures even though the network is only using CCMP. This enables an adversary, which posses the credentials to access the network, to take down the complete network by injecting two MIC failure reports every minute.

5.8 Hostapd

No vulnerability in the latest version of hostapd was discovered. Nevertheless, our impossible TKIP countermeasure attack of Section 5.2 affects hostapd v0.7.2 and earlier. Additionally, we observed that hostapd instantly deauthenticates the client if WPA1 is used and message 2 has the `Secure` bit set. Other APs would either accept this message, or silently ignore it. This unique behaviour of hostapd can be used to fingerprint and identify the implementation.

6 Conclusion

Our model-based testing technique discovered several logical bugs in implementations of the Wi-Fi handshake. Moreover, a substantial amount of these bugs are exploitable. Proof-of-concepts of selected attacks are available online [23]. Most prominently, the OpenBSD client was missing the state machine of the 4-way handshake, leading to a trivial man-in-the-middle attack against it. We consider these findings surprising, as the Wi-Fi handshake is fairly simple. Given these results, an interesting future research direction is to develop more powerful tools to (semi-)automatically detect logical vulnerabilities in implementations of network protocols.

References

[1] WiGLE: WiFi encryption over time. Retrieved 23 October 2016 from `https://wigle.net/enc-large.html`.

[2] Nadarajah Asokan, Valtteri Niemi, and Kaisa Nyberg. Man-in-the-middle in tunnelled authentication protocols. In *SPW*, 2003.

[3] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE SP*, 2015.

[4] Andrea Bittau, Mark Handley, and Joshua Lackey. The final nail in WEP's coffin. In *IEEE SP*, 2006.

[5] Sebastian Brenza, Andre Pawlowski, and Christina Pöpper. A practical investigation of identity theft vulnerabilities in eduroam. In *WiSec*, 2015.

[6] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE SP*, 2014.

[7] Laurent Butti and Julien Tinnes. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology*, 4(1):25–37, 2008.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[9] Aldo Cassola, William Robertson, Engin Kirda, and Guevara Noubir. A practical, targeted, and stealthy attack against WPA enterprise authentication. In *NDSS*, 2013.

[10] IEEE 802 LAN/MAN Standards Committee et al. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. *IEEE Standard*, 2012.

[11] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security*, 2015.

[12] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In *SAC*, Lecture Notes in Computer Science, 2001.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

[14] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[15] Changhua He and John C Mitchell. Analysis of the 802.11 i 4-Way handshake. In *Proceedings of the 3rd ACM workshop on Wireless security (WiSE)*, 2004.

[16] Hyunuk Hwang, Gyeok Jung, Kiwook Sohn, and Sangseo Park. A study on MITM (man in the middle) vulnerability in wireless network using 802.1x and EAP. In *ICISS*, 2008.

[17] IEEE Std 802.11-2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, 2012.

[18] Manuel Mendonça and Nuno Neves. Fuzzing Wi-Fi drivers to locate security vulnerabilities. In *EDCC*, 2008.

[19] Pieter Robyns, Bram Bonné, Peter Quax, and Wim Lamotte. Short paper: exploiting WPA2-enterprise vendor implementation weaknesses through challenge response oracles. In *WiSec*, 2014.

[20] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *CCS*, 2016.

[21] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *TISSEC*, 2004.

[22] Erik Tews and Martin Beck. Practical attacks against WEP and WPA. In *WiSec*, 2009.

[23] Mathy Vanhoef. Proof of concepts of attacks against wi-fi implementations. Last retrieved 17 July 2017 from `https://github.com/vanhoefm/blackhat17-pocs`.

[24] Mathy Vanhoef and Frank Piessens. Practical verification of WPA-TKIP vulnerabilities. In *ASIA CCS*, pages 427–436. ACM, 2013.

[25] Mathy Vanhoef and Frank Piessens. Advanced Wi-Fi attacks using commodity hardware. In *ACSAC*, 2014.

[26] Mathy Vanhoef, Domien Schepers, and Frank Piessens. Discovering logical vulnerabilities in the Wi-Fi handshake using model-based testing. In *ASIA CCS*. ACM, 2017.

[27] Michal Zalewsk. American fuzzy lop fuzzer. Retrieved from `lcamtuf.coredump.cx/afl/`.