Symbolic Execution of Security Protocol Implementations: Handling Cryptographic Primitives

Mathy Vanhoef
imec-DistriNet, KU Leuven
Mathy.Vanhoef@cs.kuleuven.be

Frank Piessens
imec-DistriNet, KU Leuven
Frank.Piessens@cs.kuleuven.be

Abstract

We show how to efficiently simulate cryptographic primitives during symbolic execution. This allows analysis of security protocol implementations, and revealed several flaws in implementations of WPA2's 4-way handshake.

Traditional symbolic execution engines cannot handle cryptographic primitives, because analyzing them results in complex symbolic expressions that cannot be handled by the SMT solver. We prevent this by simulating their behaviour under the Dolev-Yao model. This enables efficient symbolic execution of security protocols implementations, making it possible to detect common programming mistakes in them. We also show how to detect misuse of cryptographic primitives. That is, we can detect trivial timing side-channels, and we can identify decryption oracles where unauthenticated decrypted data influences the program's behaviour. We apply our technique on three client-side implementations of WPA2's 4-way handshake. This uncovered timing side-channels when verifying authentication tags, a denial-of-service attack, a stack-based buffer overflow, and also revealed a non-trivial decryption oracle. We confirmed all vulnerabilities in practice, and discuss them in detail.

1 Introduction

Implementing security protocols is a tedious and error-prone task. Unfortunately, history shows that any errors or bugs in a security protocol implementation can lead to devastating vulnerabilities. For example, an integer overflow in OpenSSH allowed an adversary to remotely execute arbitrary code on affected servers [13, 40]. Older versions of OpenSSH even had a vulnerability that allowed code execution on both clients and servers [12]. As another example, the Heartbleed vulnerability in OpenSSL allowed an adversary to compromise all Transport Layer Security (TLS) connections made to a vulnerable server [2, 23]. Comparable failures also occurred in other implementations of the TLS protocol [41], and in implementations of other security protocols such as Wi-Fi Protected Access two (WPA2) [51], IPsec [17], Bluetooth [14], RSA key generation algorithms [39], and so on. In all these examples, implementation issues undermined the security guarantees normally provided by the protocol. Inspired by these observations, our goal is to use symbolic execution, instead of traditional fuzzing, to find bugs in security protocol implementations.

Traditional fuzzing is used by several previous works to test security protocol implementations. Here, concrete input is fed to an implementation, after which its behaviour is monitored for irregularities. This was used to test protocols such as TLS, and revealed flaws in the state machines of several libraries [5, 19], uncovered flawed client-side X.509 certificate parsers [9, 16], and detected various buffer boundary violations [44]. However, a major downside of fuzzing is that it may not explore all code paths, meaning certain vulnerabilities remain undetected.

In contrast to fuzzing, symbolic execution attempts to exhaustively explore all code paths of a program. This is done by running the program on symbolic inputs, instead of on concrete ones. When a branch is encountered that depends on symbolic input, execution is forked, and all feasible paths are explored. This technique has been used to test a DHCP server [11], to verify properties of distributed protocols [42], and to check protocol requirements of Zeroconf and DHCP implementations [45]. However, handling security protocols that use cryptographic primitives, such as encryption and authentication, remains challenging. The main obstacle is that the cryptographic primitives in these protocols generate complex symbolic expressions, which cannot be handled by the SMT solver. We show how this problem can be avoided by simulating the behaviour of cryptographic primitives, instead of symbolically executing them.

Apart from using symbolic execution to detect common programming mistakes, we also present techniques to discover misuse of cryptographic primitives. In particular, we can detect when authentication tags are not verified in constant-time. Moreover, we can also detect decryption oracles, where an implementation's behaviour is influenced by decrypted but unauthenticated data.

To evaluate our techniques, we implement them on top of the KLEE symbolic execution engine [10], and apply our tool to three client-side implementations of WPA2's 4-way handshake. Note that this handshake is used by nearly all protected Wi-Fi networks, and provides both mutual authentication and session key agreement. This revealed several vulnerabilities. First, two clients verify authentication tags using timing-unsafe memory compares, resulting in trivial timing side-channels. Second, Intel's iwd daemon contains a denial-of-service vulnerability that can be triggered by a malicious Access Point (AP). Third, MediaTek's implementation contains a stack-based buffer overflow in code that processes decrypted data, which can be exploited by a malicious AP. Fourth, wpa_supplicant contains an non-trivial decryption oracle that is caused by processing decrypted but unauthenticated data. This decryption oracle can be exploited when the victim connects to a WPA2 network using the old TKIP encryption algorithm. It can be abused to decrypt the group key transported in message 3 of the 4-way handshake. Finally, when manually preparing MediaTek's code for symbolic execution, we also discovered that it incorrectly implemented the AES unwrap algorithm. We confirmed all vulnerabilities in practice, and will explain them in detail.

To summarize, our main contributions are:

- We modify the KLEE symbolic execution engine to efficiently handle cryptographic primitives, by simulating their behaviour under the Dolev-Yao model.
- We show how to detect timing side-channels and decryption oracles that are caused by improper use of (legacy) cryptographic primitives.
- We evaluate our technique against WPA2's 4-way handshake, and explain how all discovered vulnerabilities can be exploited in practice.

The remainder of this paper is structured as follows. Section 2 introduces symbolic execution, and relevant aspects of the 802.11 standard. Section 3 explains how we simulate cryptographic primitives, and in Section 4 we apply our technique to the 4-way handshake of WPA2. We discuss our results in Section 5, and go over related work in Section 6 . Finally, we conclude in Section 7.

2 Background

In this section we introduce symbolic execution [34], go over the main features of WPA2, and explain how the 4-way handshake works [31].

2.1 Symbolic Execution

Traditional fuzzing techniques, where random input is generated and fed to a program, are unable to explore all possible code paths within reasonable time. 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. Symbolic execution overcomes this problem by running the program not on random concrete inputs, but on symbolic inputs [34]. During such a symbolic execution, the program memory and output values are represented by symbolic expressions over the symbolic input values. Additionally, when a conditional statement is encountered, symbolic execution is forked over all feasible branches. This assures that all code paths (i.e. sequences of conditional branches) that the program might execute are analyzed. For each code path, a path constraint is maintained that defines the inputs for which the program executes along this path. Put differently, the path constraint records all conditions the symbolic input must satisfy for an execution to reach a particular location in the program. To determine whether a conditional branch is feasible, and to generate concrete inputs that follow certain code paths, a boolean satisfiability problem (SAT) solver or a Satisfiability Modulo Theories (SMT) solver is used. Several open and closed source tools implement this technique, with examples being DART [26], KLEE [10], and SAGE [27].

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 code paths in the program. We show how these limitations can be managed when symbolically executing security protocols.

2.2 Wi-Fi Protected Access

A connection to a modern protected Wi-Fi network is managed using a Robust Security Network (RSN) association. The technical features and capabilities of an RSN association are defined in the 802.11 standard [31, §12]. A subset of RSN is certified by the Wi-Fi Alliance under the more well-known name Wi-Fi Protected Access version two (WPA2). The first version, called WPA, was based on a draft of the standard. This means that WPA is almost identical to WPA2. Moreover, they both support the older (WPA-)TKIP encryption protocol, and the more modern (AES-)CCMP protocol. An AP advertises its supported encryption protocols in periodically broadcasted beacons. When connecting to a network, the client informs the AP which encryption protocol (i.e. cipher suite) it wants to use, and then executes the 4-way handshake to establish a secure connection.

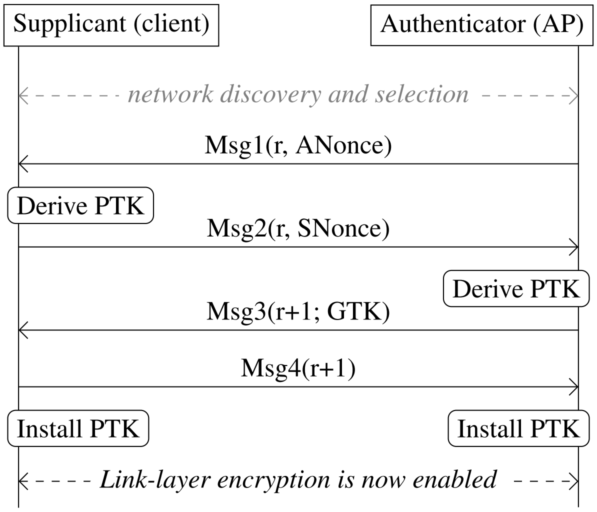


Figure 1: Simplified overview of the messages exchanged when executing the 4-way handshake between a client (supplicant) and an AP (authenticator).

<table>
<tr><td colspan="2">Protocol Version
1 byte</td><td colspan="2">Packet Type
1 byte</td><td colspan="2">Body Length
2 bytes</td></tr>
<tr><td colspan="6">Descriptor Type – 1 byte</td></tr>
<tr><td colspan="3">Key Information
2 bytes</td><td colspan="3">Key Length
2 bytes</td></tr>
<tr><td colspan="6">Key Replay Counter – 8 bytes</td></tr>
<tr><td colspan="6">Key Nonce – 32 bytes</td></tr>
<tr><td colspan="6">EAPOL Key IV – 16 bytes</td></tr>
<tr><td colspan="6">Key RSC – 8 bytes</td></tr>
<tr><td colspan="6">Reserved – 8 bytes</td></tr>
<tr><td colspan="6">Key MIC – variable</td></tr>
<tr><td colspan="3">Key Data Length
2 bytes</td><td colspan="3">Key Data
variable</td></tr>
</table>

Figure 2: Layout of EAPOL-Key frames [31, §11.6.2].

Figure 1 illustrates the messages exchanged during the 4-way handshake. During this handshake, the client is called the supplicant, and the AP is called the authenticator (we use these terms as synonyms). The AP starts by sending message 1, which contains a random number called the ANonce. This message is the only one that is not protected by a Message Integrity Code (MIC). In response, the client replies with message 2, which contains a random SNonce. Once both endpoints learn each other's nonces, they combine them with a shared secret to derive a session key called the Pairwise Transient Key (PTK). The third message is sent by the AP, and if WPA2 is used, it will contain the group key (GTK) used by the network. In case the older WPA protocol is used, the GTK is transported to the client in a separate group key handshake. Finally, the client replies with message 4, and installs the session key (PTK).

All messages in the 4-way handshake are defined using EAPOL-Key frames. The structure of these frames is shown in Figure 2. To symbolically execute implementations of the 4-way handshake, and to understand our discovered attacks, we will briefly discuss the most important fields. First, the 5-byte header defines the length of the frame and its type. This header is followed by the key information field. It contains a 3-bits key descriptor version subfield, and eight one-bit flags that called the key info flags. For us, only the MIC and encrypted flags are important. If the MIC flag is set, the authenticity of the frame is protected using a MIC. When the encrypted flag is set, the key data field is encrypted. The key descriptor field defines the cipher suite that is used for authentication and encryption. This is either AES with HMAC-SHA1, or RC4 with HMAC-MD5. The choice between these cipher suites, and hence the value of the key descriptor field, depends on the pairwise cipher selected by the client. More precisely, if CCMP is chosen, AES wrap [43] with HMAC-SHA1 is used for encryption and authentication, respectively. Otherwise, RC4 encryption with HMAC-MD5 is used. For example, when WPA2 is used, the group key is stored in the key data field of message 3, meaning it is encrypted with the PTK using either RC4 or AES wrap. The content of the key data field is encoded using Information Elements (IEs), and will be explained in more detail in Section 4.2. Finally, the replay counter field is used to detect replayed frames. The AP increments the replay counter after transmitting a new frame. When the client receives a frame, it first assures the replay counter is fresh, and it will then echo the received replay counter in its response.

Note that in Figure 1 we used the notation

$$\text{MsgN(r, Nonce; GTK)}$$

to succinctly describe messages. It represents message N of the 4-way handshake, having a replay counter of *r*, and with the given nonce (if present). All parameters after the semicolon are stored in the key data field, and hence are encrypted using the PTK.

3 Handling Cryptographic Operations

In this Section we show how to efficiently simulate cryptographic primitives during symbolic execution. Building on top of this, we also present a technique to detect misuse of legacy cryptographic primitives.

3.1 Simulating Cryptographic Primitives

Our first goal is to efficiently perform symbolic execution of security protocol implementation, in other words, of protocols that use cryptographic primitives. Consider for instance the following typical code snippet:

```
def handle_packet(session, packet):
    # Verify authenticity and decrypt packet
    data = aead_decrypt(session.key, packet)
    if data == None: return

    if data[0] == COMMAND: process_command(payload)
    # Continue processing all payload types ...
```

Here the receiver first verifies and decrypts the incoming packet. If successful, the packet's payload is parsed and processed. When using traditional symbolic execution to analyse this code, and marking the packet as symbolic, a complex symbolic expression will be assigned to the decrypted `data` variable (in a sense this is what encryption is designed to do). A consequence of this is that any evaluations on the decrypted payload result in complex queries to the SMT solver. These queries are either slow, or cannot be answered in reasonable time. For example, when using KLEE to symbolically execute ChaCha20, the resulting SMT queries took several seconds to complete. Though this is not very slow, other encryption algorithms, such as AES in either counter or CBC mode, resulted in significantly slower queries. Moreover, when symbolically executing RC4, it took too long to generate the symbolic expression of the decrypted `data` variable. All combined, this means that in practice the input processing code will not be fully explored, or may not even be reached at all. This is problematic, because many vulnerabilities are present precisely at locations where (decrypted) input is parsed and processed [36].

To enable efficient symbolic execution of code that employs cryptographic primitives, we will simulate their behaviour, and thereby avoid generating complex symbolic symbolic expressions. In particular, we will simulate encryption, decryption, and hash functions under the Dolev-Yao model [20]. That is, we assume cryptographic primitives are perfect and unbreakable.

To simulate decryption, we treat the decrypted output of a cryptographic primitive as a fresh (unconstrained) symbolic variable. We also record the relationship between the symbolic input, and the new symbolic output. This will allow us to determine the source of symbolic variables representing decrypted data. More precisely, after symbolic execution completed or was halted, the recorded relationships can be used to manually construct ciphertext that decrypts to the desired (concrete) plaintext. Although automatically constructing such ciphertexts should be possible, our tool does not support this currently. More importantly, since the decrypted output is a fresh symbolic variable, it can now easily be analyzed without resulting in complex queries to the SMT solver. Encryption is simulated in an analogous manner.

We employ a similar approach to simulate hash functions, by treating the output (i.e. the digest) as a fresh symbolic variable. As a result, complex SMT queries are avoided when the output of a hash function is compared to another (concrete or symbolic) variable. Again we record the relationship between the fresh symbolic variable representing the output, and the inputs of the hash function. Similarly, Hash-based Message Authentication Codes (HMACs) can also be simulated by treating the output as a fresh symbolic variable, and recording that this new variable is the output of an HMAC primitive.

Primitives that combine authentication and encryption (i.e. authenticated encryption) can be simulated in one step, similar to how encryption and decryption is simulated. For instance, the primitive `aead_decrypt` in our example code is simulated by treating `data` as a fresh symbolic variable, and recording that the new symbolic variable was the result of applying the authenticated decryption primitive to the symbolic `packet` variable.

3.2 Detecting Misuse of Legacy Crypto

Several protocols still support older cryptographic primitives for backwards compatibility. One common example is that authentication and encryption is provided by two different primitives. For example, encryption may be done using RC4, and authentication using an HMAC. If for some reason the implementation does not properly combine these primitives, it may be vulnerable to attacks.

We show how to detect two common misuses of cryptographic primitives. First, we can detect if an authentication tag (e.g. the output of an HMAC) is not securely compared to its expected value. Second, we can detect if there are code paths were data is decrypted but not authenticated.

To detect if an authentication tag is not securely compared to its expected value, we search for all code paths where only a few bytes of the digest are in the path constraint. This indicates that either the digest was truncated, or that it is being compared to its expected value using a timing-unsafe memory comparison. Both cases indicate the presence of a vulnerability, where the adversary can abuse timing information to guess or determine valid authentication tags.

To detect if data is decrypted but not authenticated, we first isolate code paths where such decrypted data influences the behaviour of the implementation. This is done by seeing if the output of an unauthenticated decryption primitive occurs either in the path constraint, or in the payload of a transmitted packet. In a sense, we use the fresh symbolic variables created to simulate cryptographic primitives as an information flow taint. Note that, if for some reason the decrypted data is not processed, there is no risk, since we assume the decryption primitive itself does not leak any information. For every individual code path and decrypted data, we check if there is a corresponding call to an authentication primitive (e.g. an HMAC). More precisely, it must hold that:

1. Either the full ciphertext, or the decrypted data, is used as an input of an authentication primitive.
2. The output digest of the authentication primitive must be fully present in the path constraint.

The first condition handles both encrypt-then-mac and mac-then-encrypt constructions, and assures that all decrypted bytes are authenticated. The second condition assures that the digest is correctly verified. If these conditions are not met, we know that decrypted but unauthenticated data influences the behaviour of the program, which in practice might lead to decryption oracles.

3.3 Implementation

We implemented our extensions on top of the KLEE symbolic execution engine [10], and this code is available for download [1]. In particular, we first extended KLEE with an operation to record relationships between two (possibly symbolic) variables. Then we implemented a library that can be used to simulate authenticated and non-authenticated decryption, and that simulates hash and HMAC functions. Decryption is simulated by marking the output buffer as a fresh symbolic variable using `klee_make_symbolic`, and recording the relationship between the input and output as being the result of an (authenticated) decryption primitive. Hashing primitives are simulated similarly: the output is marked as a fresh symbolic variable, and a relationship is recorded between the input of the hash function and its output.

4 Analyzing the 4-way Handshake

In this Section we explain how we applied our modified symbolic execution technique to three client-side implementations of the 4-way handshake.

4.1 Isolating the 4-way Handshake

Symbolically analyzing the 4-way handshake is a challenging use case because there are, to the best of our knowledge, no implementations that isolate the handshake code in a separate library. In contrast, other protocols such as TLS are generally implemented in a standalone library that isolates the protocol code from other aspects of a program. This makes it harder to symbolically execute implementations of the 4-way handshake, since we cannot simply initialize a library, and then feed it symbolic packets as input. Instead, we have to either symbolically execute the full client implementation, including its wireless stack, or manually isolate the 4-way handshake code into a separate library.

The first 4-way handshake implementation we investigated is the one contained in Intel's iwd deamon. This is a fairly new wireless daemon, designed to run on embedded devices [30]. Fortunately, symbolically executing the 4-way handshake in this case is straightforward. We can base ourselves on unit tests that initialize relevant parts of the client, and then tests the 4-way handshake as if it was isolated in a separate library. That is, we take an existing unit test, provide symbolic instead of concrete inputs, and then replace all cryptographic primitives with functions that symbolically simulate their behaviour.

The second client we studied is wpa_supplicant, which is the de facto Wi-Fi client on Linux and Android. Symbolically executing its 4-way handshake code is nontrivial. One option is to execute the full wpa_supplicant client under KLEE, however, this requires the simulation of large parts of the 802.11 netlink kernel interface. This would mean writing a library that symbolically simulates netlink API calls such as network interface configuration, network scanning, wireless station management, and so on. A more practical option is inspired by the success of analyzing iwd. In particular, we opted to first write test cases that initialize wpa_supplicant and test the 4-way handshake using concrete inputs. Once these unit tests were working, we used them as a basis for symbolically executing the 4-way handshake. This approach also has the advantage that it now becomes easier to analyze the 4-way handshake of wpa_supplicant using other tools. Indeed, other tools can now use our unit test as a basis to analyze the handshake without having to simulate or handle the 802.11 netlink kernel interface.

The third implementation we tested is contained in the kernel driver of MediaTek's Wi-Fi chips. Note that this driver is commonly used in routers, access points, and wireless repeaters. Given that the 4-way handshake is implemented in a kernel driver, it is again non-trivial to symbolically execute it. Nevertheless, similar to iwd and wpa_supplicant, we managed to isolate relevant code and write unit tests for it. That is, we first wrote a userland program that simulates kernel functions used by the driver, initializes global structures, and is then able to call the 4-way handshake code using concrete inputs. Once this worked, all that was required was changing the input from concrete packets to symbolic ones.

4.2 Constraining Information Elements

The key data field of 4-way handshake messages is encoded using $\langle t, \ell, v \rangle$ triples. Here t represents the type, Listing 1: Instructions executed by iwd's decrypt function when AES unwrap is used. Note that this is a simplified overview: iwd's real function is more complex.

```
1  uint8_t *eapol_decrypt_key_data(const uint8_t *kek,
2      struct eapol_key *frame, size_t *decrypted_size)
3  {
4    size_t key_data_len = ntohs(frame->key_data_len);
5    const uint8_t *key_data = frame->key_data;
6
7    size_t expected_len = key_data_len - 8;
8    uint8_t *buf = l_new(uint8_t, expected_len);
9
10   if (key_data_len < 24 || key_data_len % 8)
11     return NULL;
12   if (!aes_unwrap(kek, key_data, key_data_len, buf))
13     return NULL;
14
15   return buf;
16 }
```

Listing 2: Simplified code of MediaTek's function that search for the group key in the key data of message 3, and subsequently installs the extracted group key.

```
#define MAX_LEN_GTK   32
#define LEN_WEP64     5

BOOLEAN RTMPParseEapolKeyData(UCHAR *pKeyData,
  UCHAR KeyDataLen, UCHAR MsgType, BOOLEAN bWPA2)
{
  UCHAR GTK[MAX_LEN_GTK];

  if (bWPA2 && (MsgType == EAPOL_PAIR_MSG_3 ||
                MsgType == EAPOL_GROUP_MSG_1))
  {
    // Code locating GTK_KDE is removed for brevity
    GTK_KDE *pKdeGtk = (GTK_KDE*)pKDE->octet;
    UCHAR GTKLEN = pKDE->Len - 6;

    if (GTKLEN < LEN_WEP64)
      return FALSE;

    NdisMoveMemory(GTK, pKdeGtk->GTK, GTKLEN);
  }
  else if (!bWPA2 && MsgType == EAPOL_GROUP_MSG_1) {
    NdisMoveMemory(GTK, pKeyData, KeyDataLen);
  }

  APCliInstallSharedKey(GTK, GTKLEN);
  return TRUE;
}
```

and ℓ denotes the length of the value v in bytes. Hence, these are commonly called type-length-value triplets, and in the 802.11 standard they are called Information Elements (IEs). Certain types, such the Key Data cryptographic Encapsulation (KDE), have values that are further subdivided into their own type-length-value triplets.

When marking the whole key data field as symbolic, this causes a huge state explosion. For example, if there are n supported types, they can occur in $n!$ different orders. Additionally, each type has a symbolic length and value, causing additional state explosions.

We avoid state explosions during the parsing of the key data field by constraining the structure of the information elements (i.e. the type-length-value triplets). This is done by assigning concrete values to the fields t and ℓ, and requiring all types to occur in a fixed order. Concrete values for t and ℓ are derived from the 802.11 standard. Currently we support the two types required by WPA2, namely the RSN element, and the KDE element which contains the current group key. By supporting only these two elements, we indirectly also constrain the length of the key data field, and therefore also constrain the total length of handshake messages.

5 Discovered Vulnerabilities

In this section we first discuss common programming mistakes that were discovered, and then we discuss misuse of cryptographic primitives. We also explain in detail whether and how all vulnerabilities can be exploited.

5.1 Denial-of-Service Attacks

Against Intel's iwd daemon we discovered a denial-of-service attack. The vulnerability is located in the function `eapol_decrypt_key_data`, which is shown in Listing 1. The root cause of the flaw is on line 7: when the length of the key data field is smaller than 8, the unsigned variable `expected_len` will overflow to a large positive number. This causes the memory allocation on the next line to fail, which automatically aborts the program. In case the length of the key data field equals 8, `l_new` will be called to allocate zero bytes. This causes it to return `NULL`, resulting in NULL-deference, causing the program to terminate. Although there is a length check on line 10, this check is performed too late. We tested both attack scenario's in practice against version 0.3 of iwd. This confirmed our predictions, meaning an attacker can indeed exploit the integer underflow on line 7 as a denial-of-service attack. At the time of writing the vulnerability had been patched [33], but was not yet assigned a Common Vulnerabilities and Exposures (CVE) identifier.

5.2 Memory Corruption Vulnerabilities

Two stack-based buffer overflows were discovered in MediaTek's implementation of the 4-way handshake. They were assigned the Common Vulnerabilities and Exposures (CVE) identifier CVE-2018-14525. Both overflows occur in the function `RTMPParseEapolKeyData`, which parses the key data field of handshake messages.

Listing 2 contains a simplified version of the function. First, if WPA2 is used, the function searches for the KDE information element containing the group key. Once found, its group key is copied to the stack variable `GTK` (see line 19). Unfortunately, it is not checked whether the length of the group key is smaller or equal to `MAX_LEN_GTK`. Second, when the older WPA variant is used, the key data field contains the group key as-is. This key is copied to the stack on line 22. However, again there is no check to assure there is enough space to save the group key.

An attacker can exploit these stack-based buffer overflows by setting up a malicious WPA or WPA2 network, and tricking victims into connecting to it. During the 4-way handshake the malicious AP then sends a message 3 that triggers the buffer overflow. When using WPA this is accomplished with a large key data field, and when using WPA2 this is done by including a KDE element containing a large group key. Given that MediaTek implements the 4-way handshake in a kernel driver, successfully exploiting the buffer overflow results in full control over the victim's device.

We confirmed the vulnerability in practice against an Asus RT-AC51U router that operated as a repeater. After notifying MediaTek, they also confirmed the vulnerability, and were preparing a patch.

5.3 Timing Side-Channels

Our symbolic execution technique also revealed timing side-channels in both iwd and MediaTek's 4-way handshake. In particular, both use the timing sensitive `memcmp` function to compare the expected HMAC output with the received one. We conjecture that exploiting these side-channels is challenging in practice. This is because `memcmp` compares several bytes at once on most platforms, and even if it compares bytes one-by-one, the resulting time differences are hard to measure over a network [35]. Nevertheless, these details depend on the type of platform that the client is running on, and therefore we do treat these timing side-channels as vulnerabilities.

We remark that wpa_supplicant is not affected by timing attacks, because it uses its own timing-safe variant of `memcmp` called `os_memcmp_const`.

5.4 Decryption Oracle

We found a decryption oracle in wpa_supplicant, where the behaviour of the client is influenced by decrypted but unauthenticated data. This vulnerability has been assigned the identifier CVE-2018-14526. Listing 3 illustrates the root cause of the vulnerability. First, on line 6 the authenticity of the frame is verified when the MIC flag is set in the key information field. Then it decrypts the key data field on line 11 if both the encrypted flag is set, and if WPA2 is used, i.e., when `sm->proto` equals `WPA_PROTO_RSN`. Combined, when setting the Encryption flag but not the MIC flag, the client will decrypt the key data field without first verifying the authenticity of the frame. The function then checks whether the incoming frame represents message 1 or 3 of the 4-way handshake, and continues to process it further.

Listing 3: Function in wpa_supplicant that handles incoming EAPOL-Key frames. Here the key data field may be decrypted without first checking its authenticity.

```
1  int wpa_sm_rx_eapol(...)
2  {
3    // Code validing other fields is skipped
4
5    if ((key_info & WPA_KEY_INFO_MIC) && !peerkey &&
6        wpa_supplicant_verify_eapol_key_mic(...))
7      goto out;
8
9    if ((key_info & WPA_KEY_INFO_ENCR_KEY_DATA) &&
10       sm->proto == WPA_PROTO_RSN)
11     if (wpa_supplicant_decrypt_key_data(...))
12       goto out;
13
14   // Some code is skipped for brevity
15   if (key_info & WPA_KEY_INFO_MIC)
16     wpa_supplicant_process_3_of_4(...);
17   else
18     wpa_supplicant_process_1_of_4(...);
19 }
```

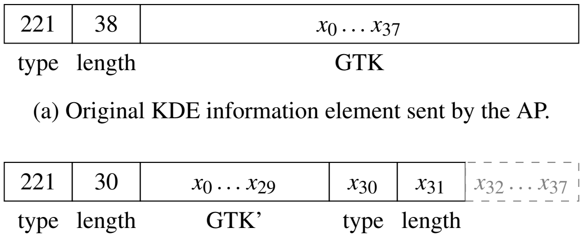
When processing either message 1 or 3, and using WPA2, the (possibly decrypted) key data field is parsed using the function shown in Listing 4. This function parses and validates the type-length-value information elements contained in the key data field. Important for us is that, depending on the content of the data, parsing may fail (see line 13-14). When parsing fails, the client aborts the handshake and disconnects from the network. When parsing succeeds, it continues with processing the frame, and replies with the next handshake message if all other checks are successful. We will abuse this difference in behaviour as a side-channel to recover encrypted information. More precisely, our goal is to decrypt the group key that the AP transports to the client in message 3 of the handshake. We will accomplish this by manipulating the encrypted key data field, and deriving plaintext values based on response(s) of the client.

Two conditions must be met before we can launch a decryption oracle attack. First, our attack is only possible if WPA2 is used. When WPA is used instead, the key data field is not decrypted due to the check on line 10 in Listing 3. Second, the client must select TKIP as the pairwise cipher. Otherwise, if the client selects CCMP, Listing 4: Function in wpa_supplicant that parses the information elements in the (decrypted) key data field.

```
/** Returns non-zero value on parsing failure */
int wpa_supplicant_parse_ies(buf, len, ie)
{
  const u8 *pos, *end;
  for (pos = buf, end = pos + len; end - pos > 1;
       pos += 2 + pos[1]) {
    /* Ignore padding */
    if (pos[0] == 0xdd && ((pos == buf + len - 1)
                           || pos[1] == 0))
      break;

    /* Key Data underflow */
    if (2 + pos[1] > end - pos)
      return -1;

    if (*pos == WLAN_EID_RSN) {
      ie->rsn_ie = pos;
      ie->rsn_ie_len = pos[1] + 2;
    } else if (*pos == WLAN_EID_MOBILITY_DOMAIN)
      // Code removed for brevity
  }

  return 0;
}
```

the AES wrap algorithm is used to protect the key data field (recall Section 2). Since AES wrap includes its own authenticity check [43], we will not able to manipulate the decrypted plaintext to construct a decryption oracle. In contrast, when the client selects TKIP, the key data field is encrypted using the RC4 stream cipher. This means an adversary can flip bits in the decrypted plaintext, by flipping the corresponding bit in the ciphertext. Interestingly, both conditions are automatically fulfilled when a Wi-Fi network uses WPA2 and only supports TKIP. After all, in that case the client must use WPA2 with TKIP. Based on a wardrive capture made in 2016 [48], roughly 20% of protected Wi-Fi networks use this configuration. Hence, in practice the decryption oracle can be exploited against 20% of protected networks.

Before we can launch our attack, and decrypt the group key contained in a captured message 3, we must let the client successfully complete the handshake. Otherwise, the client will refuse to decrypt the key data field, because a session key has not yet been negotiated. Once the client completed the 4-way handshake, an adversary can decrypt the group key transported in message 3 as follows. First, the adversary clears the MIC flag in the captured message 3. As a result, wpa_supplicant will no longer check the authenticity of the frame, but will still decrypt the key data field since the Encrpyted flag is still set. The modified frame will now be treated as a message 1 (see line 15 in Listing 3). The adversary also manipulates the encrypted key data field, in order to construct a decryption oracle that will reveal the plaintext byte-by-byte. In particular, the length field of the KDE element which contains the group key is reduced, causing the next type-length-value triplet to start in the middle of the group key (see Figure 3). We only reserve two bytes for this new type-length-value triplet, and discard any bytes after it. This effectively creates an empty information element. Parsing the decrypted key data now only succeeds if the length field of this newly created information element equals zero. This enables us to guess the plaintext byte value at the location of the length field. In particular, we guess the value of the group key at this location, and use the guessed value to flip bits in the ciphertext to set the length field to zero. If the guess was correct, parsing succeeds, and the client replies with a new message 2. If the guess was wrong, parsing fails, and the client disconnects from the network. Since the group key remains identical between different connections, the adversary can continue with guessing and trying the next value, even when the key data field is encrypted under a new session key. Once the group key has been fully recovered, it can be used to inject both broadcast and unicast traffic, and can even be used to decrypt unicast and broadcast traffic [49].



(a) Original KDE information element sent by the AP.

(b) Recovering the value of x_{31}. Parsing only succeeds if x_{31} is zero. The trailing bytes $x_{32} \dots x_{37}$ are removed from the frame before sending it to the victim.

Figure 3: Recovering the bytes $x_0 \dots x_{37}$ by shortening the KDE IE, and introducing an empty IE at the end. A concrete example recovering byte x_{31} is shown in the second figure, which can be generalized to recover any byte.

We confirmed the vulnerability against version 2.6 of wpa_supplicant on Debian 8 (“Jessie”), using the virtual mac80211_hwsim Wi-Fi driver [1]. Against this configuration, an adversary can guess a plaintext value every 14 seconds. Hence, decrypting a 16-byte group key would take 8 hours on average. To speed up the decryption process, an adversary can attack multiple clients in parallel. For instance, when attacking 16 clients in parallel, it takes less than 30 minutes to decrypt a 16-byte group key. Finally, we remark that the attack is only possible if the client accepts plaintext EAPOL-Key messages when a PTK has already been installed. Previous work has shown that several platforms meet this requirement [50].

We have notified wpa_supplicant of the vulnerability. It has been assigned the identifier CVE-2018-14526, and a patch is being worked on.

5.5 Flawed Decryption Primitives

Finally, two bugs were discovered manually, before symbolically executing any code. That is, when preparing MediaTek's cryptographic function `AES_Key_Unwrap` for symbolic execution, which implements the AES unwrap algorithm, we observed that it did not check the resulting Initialization Vector (IV). Normally this IV must equal `0xA6A6A6A6A6A6A6A6` after decryption, otherwise the decrypted data is deemed inauthentic [43]. However, the IV is never checked, meaning decrypted data is always treated as authentic.

Additionally, the AES unwrap function also does not check whether the ciphertext length is a multiple of 8 bytes. This means that if the length is not divisible by 8, trailing data smaller than 8 bytes is copied as-is to the output buffer. An adversary can abuse this to inject up to 7 plaintext bytes at the end of the ciphertext.

Recall from Section 2 that all 4-way handshake messages, except message 1, are also protected by a MIC over the full EAPOL-Key frame. This means an attacker would first have to bypass this MIC check, before they can exploit MediaTek's vulnerable AES unwrap implementation. Fortunately, this does not appear possible. This means that MediaTek's flawed AES unwrap primitive, at least when used in the 4-way handshake, is not exploitable in practice. Nevertheless, we notified MediaTek about these bugs, and they have reportedly patched them in the meantime.

6 Related Work

Several techniques exist to evaluate the correctness of network protocols. First, a protocol design can be formally modelled and its properties can be proven [8, 29]. Additionally, it is possible to create a reference implementation of the protocol in a custom language, and then prove security properties about these implementations [7, 6, 25]. The downside of this approach is that existing protocol implementations cannot be verified in this manner.

Model checking has also been used to verify properties of a protocol [21, 32, 37]. One interesting example is ASPIER, which combines software model checking with standard protocol security models to analyze properties of bounded instances of the OpenSSL handshake [15]. Another tool that analyzes implementations is CSur [28]. It models the protocol using Horn clauses, to subsequently prove certain properties. Dupressoir et al. [22] used VCC to prove memory safety and security properties of protocols written in C. Finally, static analysis can also be used to check the correctness of concrete implementations or configurations [47, 52, 24].

Symbolic execution is a general technique to discover bugs in programs [34]. Several modern tools implement this technique, examples are DART [26], KLEE [10], and SAGE [27]. Symbolic execution has two main limitations. First, certain queries to the SMT solver can be slow to practically unsolvable. Second, a state explosion might occur when analysing large programs. Aizatulin et al. [3, 4] combine symbolic execution with proof techniques, by using it to extract a ProVerif model from a protocol implementation written in C. Unfortunately, their technique is limited to protocols without branching, because it only considers a single code path. Chau et al. [16] use symbolic execution to test X.509 parsers in TLS protocols. Unfortunately, only the X.509 parser was symbolically analysed, and the implementation logic of the TLS protocol itself was not considered. Our technique to constrain information elements is similar to the one employed by Chau et al. when symbolically executing X.509 certificate parsers [16]. Finally, Corin et al. simulate cryptographic primitives during symbolic execution using rewriting rules [18]. However, their approach does not enable efficient analysis of code that processes decrypted data. In contrast, our approach does allow this.

The 4-way handshake has been extensively studied. First, He et al. proved the soundness of its design [29]. Nevertheless, it was recently shown vulnerable to key reinstallation attacks [50]. The handshake was also found to be vulnerable to certain downgrade attacks [49], and it is vulnerable to dictionary attacks [38]. Researchers also performed model-based testing of the 4-way handshake [51], and used state machine learning to discover various vulnerabilities in implementations of the 4-way handshake [46].

7 Conclusion

We successfully applied symbolic execution to client-side implementations of the 4-way handshake of WPA2, by simulating cryptographic primitives, and constraining parts of the symbolic input to prevent excessive state explosions. This revealed memory corruptions in code that processes decrypted data, uncovered insecure implementations of cryptographic primitives, and even revealed a decryption oracle. We consider this surprising, as our current symbolic execution technique is quite straightforward, yet still revealed vulnerabilities in all three tested implementations. Inspired by these results, we consider it interesting future work to further extend and improve our techniques. We also believe it is worthwhile to apply them on other security protocol implementations.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven. Mathy Vanhoef holds a Postdoctoral fellowship of the Research Foundation Flanders (FWO).

References

[1] https://github.com/vanhoefm/woot2018.

[2] Vulnerability note VU#720951: OpenSSL TLS heartbeat extension read overflow discloses sensitive information. https://www.kb.cert.org/vuls/id/720951.

[3] AIZATULIN, M., GORDON, A. D., AND JÜRJENS, J. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS)* (2011).

[4] AIZATULIN, M., GORDON, A. D., AND JÜRJENS, J. Computational verification of C protocol implementations by symbolic execution. In *Proceedings of the 19th ACM conference on Computer and communications security (CCS)* (2012).

[5] BEURDOUCHE, B., BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., KOHLWEISS, M., PIRONTI, A., STRUB, P.-Y., AND ZINZINDOHOUE, J. K. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy* (2015).

[6] BHARGAVAN, K., FOURNET, C., AND GORDON, A. D. Modular verification of security protocol code by typing. In *Proceedings of the 37th Annual Symposium on Principles of Programming Languages (POPL)* (2010).

[7] BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND TSE, S. Verified interoperable implementations of security protocols. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)* (2006).

[8] BLANCHET, B. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)* (2001).

[9] BRUBAKER, C., JANA, S., RAY, B., KHURSHID, S., AND SHMATIKOV, V. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy* (2014).

[10] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2008).

[11] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)* (2006).

[12] CERT/CC. Vulnerability note vu#945216: SSH CRC32 attack detection code contains remote integer overflow. Last retrieved 24 July 2018 from https://www.kb.cert.org/vuls/id/945216, 2001.

[13] CERT/CC. Vulnerability note vu#369347: OpenSSH vulnerabilities in challenge response handling, 2002.

[14] CERT/CC. Bluetooth implementations may not sufficiently validate elliptic curve parameters during diffie-hellman key exchange. Last retrieved 24 July 2018 from https://www.kb.cert.org/vuls/id/304725, 2018.

[15] CHAKI, S., AND DATTA, A. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE Computer Security Foundations Symposium (CSF)* (2009).

[16] CHAU, S. Y., CHOWDHURY, O., HOQUE, E., GE, H., KATE, A., NITA-ROTARU, C., AND LI, N. SymCerts: Practical symbolic execution for exposing noncompliance in X.509 certificate validation implementations. In *IEEE Symposium on Security and Privacy* (2017).

[17] CISCO. Cisco ASA software IKEv1 and IKEv2 buffer overflow vulnerability. Last retrieved 13 July 2018 from http://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20160210-asa-ike, 2016.

[18] CORIN, R., AND MANZANO, F. A. Efficient symbolic execution for analysing cryptographic protocol implementations. In *International Symposium on Engineering Secure Software and Systems (ESSoS)* (2011).

[19] DE RUITER, J., AND POLL, E. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium* (2015).

[20] DOLEV, D., AND YAO, A. C. On the security of public key protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (SFCS)* (1981).

[21] DUFLOT, M., KWIATKOWSKA, M., NORMAN, G., PARKER, D., PEYRONNET, S., PICARONNY, C., AND SPROSTON, J. *FMICS Handbook on Industrial Critical Systems*. 2010, ch. Practical Applications of Probabilistic Model Checking to Communication Protocols.

[22] DUPRESSOIR, F., GORDON, A. D., JURJENS, J., AND NAUMANN, D. A. Guiding a general-purpose C verifier to prove cryptographic protocols. In *IEEE Computer Security Foundations Symposium (CSF)* (2011).

[23] DURUMERIC, Z., LI, F., KASTEN, J., AMANN, J., BEEKMAN, J., PAYER, M., WEAVER, N., ADRIAN, D., PAXSON, V., BAILEY, M., AND HALDERMAN, J. A. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC)* (2014).

[24] FEAMSTER, N. Practical verification techniques for wide-area routing. *ACM Sigcomm Computer Communication Review 34*, 1 (2004).

[25] FOURNET, C., KOHLWEISS, M., AND STRUB, P.-Y. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS)* (2011).

[26] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementatio (PLDI)* (2005).

[27] GODEFROID, P., LEVIN, M. Y., MOLNAR, D. A., ET AL. Automated whitebox fuzz testing. In *Proceedings of Network and Distributed Systems Security (NDSS)* (2008).

[28] GOUBAULT-LARRECQ, J., AND PARRENNES, F. Cryptographic protocol analysis on real C code. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)* (2005).

[29] HE, C., SUNDARARAJAN, M., DATTA, A., DEREK, A., AND MITCHELL, J. C. A modular correctness proof of IEEE 802.11i and TLS. In *Proceedings of the 12th ACM conference on Computer and communications security (CCS)* (2005).

[30] HOLTMANN, M. New wireless daemon for linux. In *The 2nd annual systemd.conf* (2016).

[31] IEEE STD 802.11. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification*, 2016.

[32] ISLAM, S. M., SQALLI, M. H., AND KHAN, S. Modeling and formal verification of DHCP using SPIN. *Int. Journal of Computer Science and Applications (IJCSA) 3* (2006).

[33] Kenzior, D. eapol: Fix memory allocation issues. Retrieved 16 July 2018 from `https://git.kernel.org/pub/scm/network/wireless/iwd.git/commit/?id=efecce772f9a010ea70c6332de6e34bbbf5121bf`, 2018.

[34] King, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976).

[35] Mayer, D., and Sandin, J. Time trial: Racing towards practical remote timing attacks. In *Black Hat US Briefings* (2014).

[36] Momot, F., Bratus, S., Hallberg, S. M., and Patterson, M. L. The seven turrets of babel: A taxonomy of langsec errors and how to expunge them. In *IEEE Cybersecurity Development (SecDev)* (2016).

[37] Musuvathi, M., Engler, D. R., et al. Model checking large network protocol implementations. In *Proceedings of the USENIX Symposium on Network Systems Design and Implementation (NSDI)* (2004).

[38] Nakhila, O., Attiah, A., Jinz, Y., and Zoux, C. Parallel active dictionary attack on WPA2-PSK Wi-Fi networks. In *Military Communications Conference (MILCOM)* (2015).

[39] Nemec, M., Sys, M., Svenda, P., Klinec, D., and Matyas, V. The return of coppersmith's attack: Practical factorization of widely used RSA moduli. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017).

[40] OpenBSD. CVE-2002-0639: Integer overflow in sshd in OpenSSH. Last retrieved 13 July 2018 form `http://www.openssh.com/txt/preauth.adv`, 2002.

[41] Ristić, I. SSL/TLS and PKI history. Last retrieved 13 July 2018 form `https://www.feistyduck.com/ssl-tls-and-pki-history/`, 2017.

[42] Sasnauskas, R., Landsiedel, O., Alizai, M. H., Weise, C., Kowalewski, S., and Wehrle, K. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks* (2010).

[43] Schaad, J., and Housley, R. *Advanced Encryption Standard (AES) Key Wrap Algorithm*, Sept. 2002.

[44] Somorovsky, J. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 23th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016).

[45] Song, J., Cadar, C., and Pietzuch, P. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering 40*, 7 (2014).

[46] Stone, C. M., Chothia, T., and de Ruiter, J. Extending automated protocol state learning for the 802.11 4-way handshake. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (2018).

[47] Udrea, O., Lumezanu, C., and Foster, J. S. Rule-based static analysis of network protocol implementations. In *USENIX Security Symposium* (2006).

[48] Vanhoef, M. *A Security Analysis of the WPA-TKIP and TLS Security Protocols*. PhD thesis, KU Leuven, 2016.

[49] Vanhoef, M., and Piessens, F. Predicting, decrypting, and abusing WPA2/802.11 group keys. In *USENIX Security Symposium* (2016).

[50] Vanhoef, M., and Piessens, F. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017), ACM.

[51] Vanhoef, M., Schepers, D., and Piessens, F. Discovering logical vulnerabilities in the Wi-Fi handshake using model-based testing. In *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security (Asia CCS)* (2017).

[52] Wagner, D., and Dean, R. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy* (2001).