Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Model based fuzzing of the WPA3 Dragonfly handshake

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von:	Nikolai Philipp Tschacher
geboren am:	1. Mai 1991
geboren in:	Tübingen
Gutachter/innen:	Prof. Dr. Jens-Peter Redlich Prof. Dr. Björn Scheuermann

eingereicht am: .... verteidigt am: ....

Contents

Abstract. In this master thesis, a model based fuzzing framework targeting the new Simultaneous Authentication of Equals handshake (or Dragonfly) will be developed. This handshake is the central part of the WPA3 certification program, that was published in early 2018 by the non-profit organization named Wi-Fi Alliance [All19a].

This new handshake remedies two security limitations of the existing WPA/WPA2 4-way handshake: Offline dictionary attacks following an active deauthentication attack are ineffective, because Dragonfly allows only a single guess at the password by interacting with the involved handshake participants. Furthermore, since Dragonfly establishes each session a unique ephemeral Pairwise Master Key (PMK), perfect forward secrecy will be guaranteed. In addition, protected management frames will be mandatory for new devices in order to obtain WPA3 certified status by the Wi-Fi Alliance. This makes offline dictionary attacks in general infeasible.

The Dragonfly exchange does not obsolete the 4-way handshake, it merely replaces the WPA2 open system authentication in the authentication request phase with four additional frames called the SAE-Commit and SAE-Confirm message. If the confirmation token is valid, the association phase and 4-way handshake continues with the derived high entropy PMK.

A primary goal of the thesis will be a model of the finite state machine of the SAE handshake and a set of fuzzing heuristics. These are used to generate test cases designed to uncover logical vulnerabilities within SAE handshake states. Those heuristics (or rules) will tamper with the **sequence** of frames, the **content** of the frames and **format** of frames. Another goal of the thesis will be a software implementation of the model based test suite that will target soon to be expected WPA3 hardware.

Despite introducing additional security capabilities, a new version of a handshake often introduces complexity and grounds for fresh security vulnerabilities. Additionally, new hardware must coexist with old hardware during a transition period (which is often long, as history has shown in the 802.11 industry). Those facts motivate a modern model based fuzzing approach on emerging WPA3 devices and software.

Recent research has shown in the FREAK TLS/SSL downgrade vulnerability [Beu+15] that a combination of state machine composition bugs and weak cipher suites such as `RSA_EXPORT` lead to a new class of vulnerabilities. Similar methodological approaches were successful in 802.11 networks, when Vonhoef et al. demonstrated key re-installation attacks on the WPA2 4-way handshake [VP17]. Because WPA3-SAE predates the discovery of the key re-installation attacks and relies on the same EAPOL 4-way handshake, it cannot be excluded that future WPA3 devices are subject to the same attack model [wla19].

SAE authentication is fully implemented in `hostapd` and `wpa_supplicant` since version `2.7`. While these implementations will be primarily tested for deviant traces and logical state flaws, a secondary focus will be cast on unsafe programming practices which may lead to memory corruption issues such as stack overflows, heap overflows, integer overflows, format string vulnerabilities, NULL pointer dereferences and so on.

The model based fuzzer should be protocol aware, which means that it must be able to execute tests as a function of the state of network participants within a 802.11 network. In practice, this means that the fuzzer should be able to automatically cover the most important state-combinations of the new SAE/Dragonfly protocol.

1 Introduction

Wi-Fi Protected Access (WPA) is a set of security protocols and certifications proposed by the Wi-Fi Alliance to secure IEEE 802.11 networks. WPA (2003) and WPA2 (2004) were created in order to obsolete the insecure **Wired Equivalent Privacy (WEP)** standard.

In 2018, the third revision of WPA was released to the public. The core ingredient of WPA3 is an key-exchange handshake plugged in front of the old 4-way handshake. This additional handshake, which was originally standardized in 2011 [Hie+10], adds two security properties to Wi-Fi's WPA:

1. **Forward secrecy**. Attackers cannot decrypt traffic with old keys, because the negotiated keys are updated in every new instance of the handshake.
2. **Offline dictionary attack resistance**. Attackers can merely launch online attacks against the handshake. Put differently, the number of guesses from a brute force attack grows linearly with the number of authentication attempts, which can easily be regulated by the authenticator.

The security of this Simultaneous Authentication of Equals (SAE) handshake, an instance of the Dragonfly family, will be the focus of this master thesis. In particular, the main objective is the development of a model based fuzzing framework targeting WPA3 capable hardware and software.

1.1 Motivation

As soon as a new security standard is announced to the public, scientists and vulnerability researchers often anticipate an initial decrease in overall security. The reason is the introduction of untested software building blocks that frequently lead to new programming mistakes. This initial decrease in security is assumed to be not different in upcoming hardware and software with WPA3-SAE support.

Even though there is limited amount of hardware that *speaks* WPA3-SAE as of July 2019 [AVM19], it is still possible to test already existing software implementations of WPA3-SAE in widely used open source projects such as `hostapd`, `wpa_supplicant` and `iwd`.

It is straightforward to create a software access point with WPA3-SAE support and a connecting supplicant that initiates the handshake. This makes it possible to develop and test Wi-Fi fuzzers under laboratory conditions and later employ them in proprietary hardware environments that support WPA3-SAE out of the box.

The Wi-Fi industry is assumed to introduce support for WPA3-SAE in hardware on a large scale starting from 2019, because manufactures are pushed to adopt the new WPA3-SAE certification for marketing reasons and to avoid negative valence. It is of secondary nature that the old WPA2 certification is still considered to be secure from a technical standpoint, as long as high-entropy passwords are used. Those reasons motivate the development of a fuzzing framework that is capable of targeting the soon to be expected WPA-SAE hardware over the radio.

Put differently, the main motivation to subject the WPA3-SAE handshake to a fuzzing based security audit is the novelty of the protocol and the very likely widespread hardware adoption in the near future.

1.2 Objectives

The contribution of this master thesis is twofold. One objective is to follow a systematic, reproducible process of deriving a set of fuzzing test cases that target the WPA3-SAE handshake. This process can be abstracted and applied to other Wi-Fi protocol additions such as the Fast Initial Link Setup (FILS) handshake. The other objective is a practical contribution in the form of a fuzzing framework that targets existing WPA3-SAE software and hardware.

The security of WPA3-SAE implementations can be tested by in-process fuzzing and remote fuzzing. While **in-process fuzzing** focuses on a single target of evaluation and has a constant implementation overhead, it is the most effective strategy to fuzz large open source WPA3-SAE software projects such as `hostap` or `iwd`.

WPA3 capable routers and modems emerging on the hardware market require a **blackbox remote fuzzing strategy**, because it cannot be guaranteed that information about system internals can be obtained. For example, the only practical connection to a Internet of things device might be via Wi-Fi radio. For those devices, blackbox fuzzing over the radio is the only economically viable approach. It is simply not feasible to submit a myriad of WPA3 devices to a whitebox security audit.

This motivates the **dualistic fuzzing strategy** employed throughout this thesis. Additionally, at the time of writing (July 2019), there were simply not enough WPA3 capable devices on the hardware market to justify a solely remote blackbox fuzzing based approach.

This master thesis should provide room for manual security research efforts. Put differently, if a vulnerability is discovered during practical efforts of implementing the fuzzing framework, it should be included in the results section of this thesis.

The remainder of this thesis is structured into the following chapters:

Various theoretical fuzzing approaches are introduced in the second part of this introduction. Then, the related academic work is quickly outlined. In the next chapter, cryptographic fundamentals such as the Discrete Logarithm Problem and Elliptic Curve Cryptography are theoretically introduced.

In the third chapter, the WPA3-SAE handshake is thoroughly treated. The focus lies on the logic behind the two messages in the handshake as well as cryptographic aspects of the handshake. The chapter concludes with a quick summary of security issues extracted from recent research.

In the fourth chapter, practical aspects of the fuzzing environment are presented. The chapter's main purpose is to give hands-on instructions on how to replicate the laboratory setup that was used to obtain the results of this thesis.

The WPA3-SAE model is thoroughly examined in the fifth chapter. The message framing of the SAE handshake is investigated in detail. The chapter concludes by presenting a model of the SAE handshake and a table of the derived fuzzing test cases.

The last chapter presents the results obtained from the dualistic fuzzing strategy used throughout this thesis. In the second part of the last chapter, a discussion is held about the various problematic aspects of a fuzzing campaign conducted in previously unchartered WPA3 territory.

1.3 WPA3

The key idea behind the development of WPA3 was to improve the security of the WPA2-PSK handshake. However, the WPA3 development process was shielded from the public, such that independent researchers could not peer-review the newly introduced features [VR19]. Vanhoef et al. criticize that the security guarantees of the handshake are unclear, because a close variant of the Dragonfly handshake received bad reviews during the standardization [VR19].

WPA3 is a certification, which means that no new protocols are defined. WPA3 merely mandates which existing protocols devices must support. WPA3 allows a transition mode, where both WPA2 and WPA3 are simultaneously supported in order to guarantee backwards compatibility [VR19]. Furthermore, WPA3 was designed for two distinct types of networks: Normal home networks, where users authenticate with a pre-shared password and enterprise networks with more advanced authentication methods such as smart cards or certificates.

This thesis follows the notation of Vanhoef et al. and uses the term **WPA3-SAE** in order to refer to home networks (**WPA3-Personal**). WPA3-SAE mandates support for the Simultaneous Authentication of Equals handshake, which is a **Password Authentication Key Exchange (PAKE)**, meaning that authentication is performed based on a password which is shared among all handshake participants [VR19].

The output of WPA3-SAE authentication is a high-entropy **Pairwise Master Key (PMK)**, which subsequently is used as input for the 4-way handshake to derive a **Pairwise Transient Key (PTK)**. Therefore, WPA3 always involves two handshakes: The password authentication SAE handshake followed by the well established 4-way handshake. By using the high entropy PMK, the 4-way handshake is not vulnerable against dictionary attacks [VR19].

WPA3-SAE additionally enforces **Management Frame Protection (MFP)**, which is standardized in the IEEE 802.11w amendment. While WPA3 is still technically vulnerable against an offline dictionary attack, the brute force of a cryptographically random 32-byte PMK is practically not feasible. Furthermore, even if the PSK is disclosed, WPA3-SAE guarantees that it is impossible to derive the PMK and thus neutralize the encryption provided by the 4-way handshake. This security guarantee is also known as forward secrecy. [VR19]

Existing 802.11 hardware may not receive support for SAE or MFP and thus cannot obtain WPA3 certification. In order to still support those devices, the WPA3 certification defined a solution where a network simultaneously supports WPA2 and WPA3. In this transition mode, older WPA2 stations can connect using the 4-way handshake without MFP and newer stations connect using the SAE handshake with MFP [VR19].

The Dragonfly handshake was first introduced by Daniel Harkins in 2008 and was added to the 802.11 standard in 2011 [Har08]. Harkins published his reference implementation of WPA3-SAE on sourceforge [Har19c]. Vanhoef et al. note that the term **Dragonfly** refers to a complete family of PAKE handshakes, whereas SAE refers to the concrete handshake exclusively used in WPA3 and standardized in IEEE 802.11 Std 2016 [IEE16].

The SAE handshake supports Finite Field Cryptography (FFC) using multiplicative groups modulo a prime p as well as Elliptic Curve Cryptography (ECC) using elliptic curve groups modulo a prime p. The 802.11 standard enforces the support of elliptic curve NIST P-256, when SAE support is advertised [VR19]. However, support for multiplicative groups is not obligatory, as a consequence most implementations only implement elliptic curves [VR19].

At the beginning of the WPA3-SAE handshake, the user's password is encoded into a group element. In the case of ECC, the password is transformed with a hash-to-curve algorithm into a so-called *password element*. The handshake includes the exchange of two messages: A commit frame and a confirm frame. The handshake can be initiated by both participants simultaneously. In infrastructure mode, the supplicant will initiate the handshake by sending its commit frame. Then the supplicant waits for the commit and confirm frame of the authenticator. Finally, the client will send its confirm frame, thus completing the handshake.

In the commit phase, each client picks two random numbers and computes a public group element based on this number. This group element and the sum of the two random numbers are exchanged between the participants. In the confirm phase, each participant computes the shared secret and takes the x-coordinate of the resulting point. In a final step, a HMAC is calculated over all relevant handshake parameters. The result of this hash is exchanged between the participants. Upon reception of this confirm token, the recipient verifies its value, which must be possible because each party knows all relevant parameters. If the confirm frame is not verified, the handshake eventually times out. [VR19]

The WPA3-SAE handshake uses a try-and-increment method to derive a valid curve point from the pre-shared password, MAC address of all participants and counter. The hash of those inputs is used as a x-coordinate and it is tested if there is a solution for y over the elliptic curve equation (An introduction into elliptic curve cryptography is given in section 2.2)

$$y^2 = x^3 + ax + b \text{ mod p}$$

If a solution exists, the point (x, y) becomes the password element, if not, the counter is increased and another attempt is made. If the algorithm stops the iteration after a valid point was discovered, attackers could remotely measure the runtime of the handshake and conclude which program path was taken based on different response times. To prevent such side-channel attacks, a static number of iterations is used in the password element derivation algorithm ($k = 40$).

Recent research by Vanhoef et al. has shown that timing side-channel attacks were very successful against the SAE handshake. Vanhoef et al. note that the exclusion of the MAC address from the hash-to-curve algorithm would have enabled to generate the curve point in offline fashion, such that the password element can be reused among connection attempts. Without this costly derivation of the password element at the beginning of each handshake, there would be diminished attack surface for DoS attacks without losing any provided security guarantees. Furthermore, the MAC addresses are not necessary for entropy reasons and thus redundant, because random numbers are used in the hash-to-curve algorithm. [VR19]

1.4 Fuzzing Methodology

In the following section, various fuzzing methods and strategies are introduced. The section concludes with a selection of fuzzing methods used throughout this thesis.

Software testing is a well known and often employed strategy in software engineering to develop robust programs. There exist different classes of testing: **Unit tests**, where the fundamental components of the software such as functions and classes are undergoing testing scenarios. **Integration tests** examine the functionality of a collection of units in their interaction. Integration tests usually build on top of unit tests and make a statement about the correctness of the tested units in interaction. **Functional tests** take the advertised functionality of the software as a template and confirm its correctness. Functional tests verify the requirements of the end user and unit tests check the correctness of low level software units such as functions and classes. [AO16]

Fuzzing however is a testing strategy whose intention is to uncover security vulnerabilities in the software under test. It is the process of *"repeatedly running a program* *with generated inputs that may be syntactically or semantically malformed"* [Man+18]. A possible definition of fuzzing as Valentin at al. understands it is *"the execution of the program under test using inputs sampled from an input space that protrudes the expected input space of the program under test"* [Man+18]. It follows that a *fuzzer* is a program that performs fuzzing in software under test [Man+18].

One of the first mentions of **fuzzing (fuzz)** in academia was a university class taught by Barton Miller in 1988 after discovering that UNIX programs frequently crashed when they were fed random, unexpected inputs [MFS90]. The lab assignment was to *"create random inputs, and see if they break things"* [MFS90]. Their efforts revealed that one third of all UNIX utilities had issues dealing with random, unexpected inputs. [Zel+19a].

Fuzzing is an automated process intended to uncover programming mistakes in programs written in memory unsafe languages such as C or C++. Examples for programming mistakes are memory leaks, segmentation faults, stack and heap overflows, off-by-one errors, integer overflows, format string vulnerabilities and so on. While many memory safe programming languages have been proposed such as Rust, Go or Haskell, most software in widespread use is still developed in C and C++. Examples for large software projects written in C and C++ are the chromium browser, the Linux kernel, many daemons and servers used in the Internet such as apache2, nginx or bind9. Translating this huge codebase to memory safe languages is an unfeasible task in the foreseeable future. For this reason, intelligent fuzzing is one of the most lucrative strategies to uncover security vulnerabilities in an automated manner.

The intuitive strategy in fuzzing is to create inputs that are structured enough to reach a large code coverage of the program under test, while simultaneously being deviant enough from a valid message to trigger crashes.

Modern Fuzzing The taxonomy of most modern fuzzing engines is composed of three groups: Blackbox fuzzing, coverage-guided greybox-fuzzing and symbolic execution-based whitebox fuzzing.

In **blackbox fuzzing**, the logic and internal behavior of the fuzzed program is largely unknown. A blackbox fuzzer merely observes the input/output behavior of the program under test, thus treating the targeted software as blackbox [Man+18]. An example for black-box fuzzing test is the generation of a large corpus of fuzzed *jpeg* files and uploading them to an arbitrary web jpeg compression service and observing if the service crashes in an unexpected way, typically revealed by 500 internal server error responses [BPR17]. Most traditional fuzzers have been blackbox fuzzers [Man+18].

Coverage-guided greybox-fuzzing (CGF) uses lightweight binary program instrumentation to trace the code coverage reached by fuzzed input mutations [Clu19]. Greybox fuzzers typically obtain limited information about the internals and semantics of the program under test, such as performing lightweight static analysis or collecting dynamic information about code coverage by instrumenting the code at compile time[Man+18].

In order to instrument programs, greybox fuzzing engines inject few code instructions right after every conditional jump. Those code instructions are called *trampolines* and their purpose is to assign a unique identifier to the current branch and increment a coarse counter belonging to the branch. The counter is implemented as probabilistic data structure such as a count-min sketch or Bloom filter. This instrumentation enables the fuzzer to keep track of what branches are how often executed. The instrumentation is applied at compile-time to the program under evaluation. [Zel+19b]

When the greybox fuzzing engine learns that a specifically mutated seed input explored previously unknown code paths, it adds the modified seed to an growing seed corpus [Zel+19b]. In other words, greybox fuzzers leverage coverage feedback information to find new inputs that reach deeper into the program [Zel+19b]. CGF does not require manual program analysis, thus being more scalable and parallelizable compared to other whitebox fuzzing strategies [BPR17]. Put differently, CGF fuzzing engines combine various powerful concepts to yield efficient fuzzing campaigns. Table 1 explains those powerful characteristics of greybox fuzzing in-depth.

Keyword	**Concept Description**
Code coverage	Code coverage is the strategy of keeping track which statements/code blocks or branches of the program under test have been executed by the fuzzing engine. [Zel+19b]
Mutations	CGF fuzzing engines fuzz programs using a corpus of valid input data that the program under test accepts. The engine mutates random bytes in the input corpus and retains the mutations leading to an increased code coverage. The engine is said to be *guided* by increasing code coverage. Examples for mutations is the randomized swapping of bytes, deletion and insertion of a random byte and so on. [Zel+19b]
Greybox	Greybox refers to the hybrid strategy of combining blackbox and whitebox fuzzing techniques. The whitebox part encompasses the collection of code coverage information of the program under test. The mutated inputs that achieve the greatest code coverage are added to the seed corpus. CGF fuzzing engines have also aspects of blackbox fuzzing, because it is highly unlikely that every statement and internal state is tested, as it is the case with concolic execution and symbolic execution in whitebox fuzzing. [Zel+19b]

Keyword	**Concept Description**
Power Schedules	A well known limitation in coverage-guided greybox fuzzers is the exercise of certain paths with high frequency which limits overall path discovery. Power schedules distribute fuzzing time among the seeds in the fuzzing population [Zel+19b]. The goal is to maximize the time spent fuzzing seeds that lead to increased coverage in shorter time. The likelihood with which a seed is chosen from the population is the seeds *energy*. The fuzzers power schedule determines the seeds energy. A common chosen strategy is to use a exponential power schedule, that assigns the seeds energy $E(s)$ as $$E(s) = \frac{1}{F(s,p)^a}$$ where F is the number of times the path p was exercised with inputs generated from s. a is a parameterized exponent. Under a exponential power schedule, most energy is assigned to the seed that exercises the lowest-frequency path. [Zel+19b] `AFLFast` for example assigns low energy to seeds that exercise high-frequency paths, whereas seeds that exercise low-frequency paths are assigned high energy. Power schedules are therefore an instrument that prioritizes seeds in order of their likely progressiveness in terms of code coverage, allowing to search the best seeds early on. This improves the efficiency of the fuzzing engine, not it's effectiveness. [BPR17]
Directed	Directed greybox fuzzing is the idea of directing the fuzzing engine towards an assumed problematic code block, thus cutting off uninteresting branches. The directedness of the fuzzing engine is achieved by gradually assigning more energy to fuzzing seeds that are closer to the code block of interest. Despite directedness being usually a characteristic of symbolic whitebox fuzzers such as KLEE [CDE+08], Boehme et al. showed that greybox fuzzing engines can outperform symbolic execution engines. [Böh+17]
Sanitizers	The program under test is instrumented with various low level sanitizers that detect programming errors. Sanitizers are applied during compilation and linking of the program. For example, `libFuzzer` may be compiled with AddressSanitizer and MemorySanitizer as part of the LLVM compiler toolchain.

Table 1: Key concepts of coverage guided greybox fuzzing as implemented in `AFL` or `libFuzzer`.

Symbolic execution-based whitebox fuzzing is the process of symbolically executing a program under test to automatically generate test inputs [CDE+08]. The symbolic input is initially allowed to be anything. Program variables are replaced by symbolic values and concrete program operations are substituted with ones which modify symbolic values [CDE+08]. Whenever a program branches based on a condition, the symbolic execution engine follows both branches and creates a path constraint named *path condition*. The path condition is the manifestation of the symbolic values that lead to a certain point in the program. If the symbolic execution encounters a bug, a test case is automatically generated using a SMT solver by solving the path condition and obtaining concrete values that trigger the bug. Symbolic execution engines work under the assumption that the code in question is deterministic. [CDE+08]

In general, white box fuzzing generate test cases by analyzing the internal state of the program under test, which makes the overhead of whitebox fuzzing much higher compared to blackbox fuzzing [Man+18].

There exist many different open source fuzzing engines for the different fuzzing strategies introduced above. Examples for modern coverage-guided greybox fuzzing engines are AFL (american fuzzy lop)[1] and `libFuzzer` (part of the LLVM toolchain)[2]. AFL and `libFuzzer` use genetic algorithms to mutate inputs in order to increase code coverage. The LLVM toolchain with the C compiler `clang` is used to compile the fuzzing target and to add a wide range of sanitizers such as AddressSanitizer, LeakSanitizer, MemorySanitizer or UndefinedBehaviorSanitizer for detection of different classes of programming mistakes [Clu19][3].

AddressSanitizer is responsible for detecting various memory errors such as use after free (dangling pointer) bugs, heap/stack/global buffer overflows. MemorySanitizer detects the reading of uninitialized memory in C/C++ software, before the memory was written. UndefinedBehaviorSanitizer[4] detects a wide range of undefined behavior in the C/C++ programming language family such as performing pointer arithmetic that overflows, signed integer overflows, where the result of a signed integer computation cannot be represented in its type and even unsigned integer overflows, which often occur unintentionally as the DoS vulnerability in section 6.2 demonstrates.

LLVM also provides the functionality for coverage instrumentation that grey-box fuzzers require [Clu19]. This in-process fuzzing strategy can be combined with a initial input corpus which increases the likelihood that a large code coverage is reached quickly.

An example for a symbolic execution built on top of the LLVM compiler toolchain is KLEE[5], initially developed by Stanford scientist and published in a widely regarded paper about the symbolic execution of GNU coreutils [CDE+08].

[1] AFL, accessed on 4th August 2019, `http://lcamtuf.coredump.cx/afl/`

[2] libFuzzer, accessed on 4th August 2019, `http://llvm.org/docs/LibFuzzer.html`

[3] Sanitizers, accessed on 7th August 2019, `https://github.com/google/sanitizers`

[4] UndefinedBehaviorSanitizer, accessed on 7th August 2019, `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

[5] KLEE, accessed on 4th August 2019, `http://klee.github.io/`

The company Google continuously fuzzes large open source software projects with its cloud based fuzzing infrastructure (mainly with the fuzzing engine `libFuzzer`) OSS-Fuzz [OSS19]. Google provides monetary incentives for software maintainers to integrate their libraries into the cloud fuzzing architecture. Over 11 thousand bugs have been found as of August 2018 [Clu19].

For example, fuzzing a proprietary 802.11 router over the air is considered black-box fuzzing, whereas the compilation of a image parsing C library with the LLVM toolchain including `libFuzzer` and AddressSanitizer is grey-box fuzzing.

1.4.1 Fuzzing Strategy

The title of this thesis advertises a **model based** fuzzing approach. Model based fuzzing as this thesis understands it, is protocol and state aware fuzzing. It is an improvement over dumb blackbox fuzzing techniques, where the system of interest is fed randomly generated input mutations without being aware of the internal state of the target.

A traditional black-box fuzzing approach has many downsides. For example, when the fuzzer creates an Auth-Commit frame in the WPA3-SAE handshake and choses a randomly fuzzed value for the public group element, the probability of selecting a valid group element is extremely low. Therefore, the if statement that confirms the validity of the element prevents the frame from reaching a potentially vulnerable section.

There are two fundamentally different fuzzing strategies employed during this thesis. **Remote fuzzing** and **in-process (in-memory) fuzzing**. Remote fuzzing refers to the process of manufacturing fuzzed frames, sending them over the air to the targeted 802.11 station and observing potential effects.

In-process fuzzing on the other side targets a suitable single function in the program under test, compiling it using a coverage guided greybox fuzzing engine such as `libFuzzer`, creating a corpus and then running the fuzzer with the corpus as input.

During this thesis, both fuzzing strategies will be used. An advantage of in-process fuzzing is the speed and availability of mature open source toolchains (such as `AFL` and `libFuzzer`). An disadvantage is that only a selected function is targeted. Additionally, in complex Wi-Fi software, it is often not easy to identify a fuzzable function that handles a broad range of tainted data. Put differently, greybox fuzzing comes with the practical problem of finding a single function that accepts remotely controllable data. The setup of such a fuzzing driver requires constant work overhead for each fuzzed function interface. The implementation of such fuzzing drivers is complex, because the internal state of the software needs to be replicated. Due to such practical difficulties, the Google launched OSS-Fuzz project rewards large open source projects up to $20,000 for ideal integration into the distributed cloud fuzzing architecture [Cha+19].

An advantage of remote fuzzing is the fact that the complex interplay of 802.11 software as a whole is not lost. Often, security vulnerabilities manifest exactly in this interplay of different layers of abstraction. Once a remote fuzzer is functional, applying it to new proprietary supplicants does not produce additional work.

1.5 Related Work

In order to uncover security vulnerabilities in upcoming devices with WPA3-SAE capabilities, it is crucial to understand the most recent research about 802.11 security and previous fuzzing attempts.

A large part of the motivation to use a model based approach in this thesis was Vanhoef's Paper *Discovering Logical Vulnerabilities in the Wi-Fi Handshake Using Model-Based Testing* [VSP17] and Beurdouche et al. paper *A Messy State of the Union: Taming the Composite State Machines of TLS* [Beu+15]. Their research about vulnerabilities in the WPA 4-way handshake and TLS handshake inspired the model based fuzzing approach of the SAE handshake in this thesis.

A main effort of this thesis will be the understanding of previous work of 802.11 fuzzing attempts. Major contributions have been made by Keil/Kolbitsch [KK07], Butti [BT08], Mendonça [MN08]. Especially Butti with his open source model based fuzzer wifuzzit[6] was very successful in uncovering many low level programming security vulnerabilities and served as inspiration for the creation of `dragonfuzz.py` contributed by this thesis.

There was an explosion in research regarding Wi-Fi fuzzers in the years from 2006 to 2008. After that, not many researches followed a solely fuzzed based approach. An exception is Vanhoef (2017) who used symbolic execution to fuzz the 4-way handshake [VP18b]. Another fuzzing based security research was used by Chen et al. in their work on fuzzing IoT devices with the corresponding IoT app as a fuzzing entry point [Che+18]. A big advantage of using the official app as an fuzzing entry point is the saved time that would otherwise go in reverse engineering the protocol or firmware.

However, most security research in the 802.11 field focused on more manual research techniques, such as the key reinstallation attack that exploits implementation flaws in cryptographic key handling during the 4-way handshake [VP17] and a follow up paper that shows the existence of similar attacks in the Fast Initial Link Setup (FILS) and Tunneled directlink setup PeerKey (TPK) handshakes [VP18a].

Interestingly enough, the research that lead to the heavily publicized key reinstallation vulnerability was inspired by general work on paper that researches logical vulnerabilities in the 4-way handshake [VSP17].

Vanhoef et al. follow a model based technique that generates a set of representative test cases which cover all states of the 802.11 handshake. The test cases cover various edge cases in each state [VSP17]. If the test case triggered an anomaly is verified manually.

[6] wifuzzit, accessed on 4th August 2019, `https://github.com/0xd012/wifuzzit`

The work from Vanhoef et al. is inspired by [DP15]. Beurdouche et al. constructed a simplified model of the TLS handshake. Based on this model, they constructed so-called deviant traces. Traces are sequences of messages that should not be accepted by a secure implementation of TLS [VSP17]. This technique is commonly called *protocol state fuzzing* and was used to find logical vulnerabilities in TLS [Adr+15] [Beu+15] [DP15]

Vanhoef et al. apply test generation rules from a model of the 802.11 handshake to generate a set of test cases. *"A test case is essentially a sequence of messages to be transmitted to the authenticator, the expected replies, and a method to determine whether this exchange resulted in a successful connection or not."* [VSP17]. Vanhoef states that the step of defining appropriate set of test generation rules is the crucial point of their analysis. Their test generation rules are more or less similar to protocol aware fuzzing, with the exception that the overall space of sets is much smaller compared to traditional fuzzing and that no random numbers are used which makes test cases deterministic. [VSP17]

This transition in research indicates a general development in the security field. As long as no thorough black box fuzzing was applied to the software in question, it is relatively easy to find low hanging programming mistakes. On the other hand, creating a model aware 802.11 fuzzer that ventures deep into code paths without indefinite resources is nearly impossible.

As the research indicates, there are several hard to overcome limitations in black-box fuzzing:

1. As soon as protocols become stateful and more complex through authentication and assocation, it is very hard to draw conclusions when and where something went wrong.

2. It is hard to setup a hardware environment for fuzzing. This struggle may be somewhat mediated with virtual machines / simulation software.

3. It is practically impossible to trigger logical programming mistakes with dull blackbox fuzzing techniques.

4. It is hard to assemble a diverse enough collection of devices with many different chipsets and 802.11 firmware to obtain an large testing corpus.

A recent paper from Vanhoef and Ronen published in April 2019 stands out among other papers, because it directly relates to the topic of this thesis: *Dragonblood: A Security Analysis of WPA3's SAE Handshake* [VR19]. The most profound contributions from their work will be covered here and the parts that advance this thesis will be highlighted.

Vanhoef et al. state in their research that the SAE handshake is affected by several different design flaws, such as a *password partitioning attack* that makes use of **timing and cache-based side-channel leaks**. Those cache-based side-channel attacks exploit the hash-to-curve algorithm of SAE (referred to as *hunting an pecking technique* in the Dragonfly RFC [Har15]). Vanhoef et al. state that brute forcing all 8-character lowercase passwords takes less than 125$ of Amazon computing resources. Therefore, the attack circumvents exactly the purpose of Dragonfly: The intractability to launch an offline dictionary attack.

One novelty of WPA3 is a transition mode, where WPA2 and WPA3 are simultaneously supported for backward compatibility [VR19]. Vanhoef et al. demonstrate the following issues in WPA3:

1. The anti-clogging mechanism of WPA3-SAE is not able to prevent denial-of-service attacks. The authors show a way to overload the CPU of a professional access point.

2. When operating in transition mode, a dictionary attack against WPA3 is possible by downgrading clients to WPA2. Even though the 4-way handshake detects the downgrade and aborts, the frames exchanged during the aborted 4-way handshake are sufficient to launch on offline dictionary attack.

3. They also present a novel micro-architectural cache-based side channel attack against the SAE handshake that leaks information about the password used, even though the hash-to-curve algorithm already included countermeasures against such attacks.

4. They furthermore demonstrate how the recovered timing and cache info is used to perform an offline password partitioning attack.

[VR19]

2 Cryptographic Fundamentals

The introduction to relevant cryptographic primitives used in WPA3 will be discussed in this chapter. This cryptographic intro is primarily based on the book *Understanding Cryptography: A Textbook for Students and Practitioners* [PP09] and *Introduction to modern cryptography* [LK14]. In order to create a powerful WPA3 fuzzing framework, the required cryptographic primitives must be correctly implemented and theoretically introduced.

The cryptographic primitives used in the WPA3 SAE handshake are mainly based on Finite Field Cryptography (FFC) using multiplicative groups modulo a prime and Elliptic Curve Cryptography (ECC) using elliptic curve groups modulo a prime [Har15]. Both primitives are based on the discrete logarithm problem, on top of which the public key cryptosystems are constructed. 802.11 stations that advertise support for SAE must implement the elliptic curve NIST P-256, whereas support for other groups is optional. Put differently, this means that there is no mandated support for MODP groups [VR19]. Therefore, most SAE implementations support only elliptic curves [VR19].

The WPA3 SAE handshake is a balanced Password Authentication Key Exchange (PAKE) protocol. PAKE protocols provide strong security using weak passwords. This property is obtained by allowing guesses at the password only during direct interaction. Typically, the input for a PAKE protocol is a pre-shared secret and the output is a Pairwise Master Key (PMK). [HR10]

2.1 Public-key Cryptosystems based on the Discrete Logarithm Problem

A well known public-key scheme is RSA, which is based on the hardness of factoring large integers. The factorization of large integers is the one-way function on top of which RSA is constructed. A function is said to be one-way, if it is computationally easy to compute $f(x) = y$ but hard to find the inverse $f^{-1}(y) = x$.

SAE builds on top of another widely used one-way function that can be used to build asymmetric crypto schemes: The **discrete logarithm problem (DLP)**. Many cryptographic schemes rely on the computational intractability of solving discrete logarithms. The Diffie-Hellman key exchange is probably the first protocol that comes to mind when the DLP is introduced. Another protocol that uses the DLP internally is the Digital Signature Algorithm (DSA), a signature algorithm that is widely deployed in Public Key Infrastructure. [PP09]

In the following sections, the Diffie-Hellman protocol is presented as well as the algebraic structures behind the DLP: Cyclic multiplicative groups. Then, the most common ideas behind popular attacks on the DLP are introduced. The motivation for this cryptographic introduction is the fact that the SAE handshake is more or less a modified

Diffie-Hellman key exchange. Having a solid understanding of the primitives will help finding potential security issues in WPA3.

Diffie-Hellman Key Exchange The Diffie-Hellman Key Exchange (DHKE) was proposed by Whitfield Diffie and Martin Hellman in 1976 and was the first published asymmetric crypto scheme [PP09]. The Diffie-Hellman Key Exchange solves the well known problem of sharing a secret key between two parties by communicating over an insecure channel. The DHKE is widely used in many important cryptographic protocols such as SSH, TLS and IPSec [PP09].

The fundamental idea behind the DHKE is that exponentiation in Z_p^* with p prime is a commutative one-way function [PP09].

$$k = (\alpha^x)^y \equiv (\alpha^y)^x \text{ mod p}$$

When the key k is computed according to the equation above, k can be used as a session key between the two handshake participants.

The DHKE consists of two protocols, the set-up protocol and the main protocol. The task of the set-up protocol is to choose domain parameters. Each party chooses a large prime p, an integer $\alpha \in \{2, 3, ..., p-2\}$ and proceeds to publish p and α.

As soon as both parties (referred to as Alice and Bob, as it is convention in cryptography) have established their domain parameters, they can generate a joint secret key k with the following key-exchange protocol:

Alice chooses a private key a and computes a public key A as follows:

$$\begin{aligned} a &\in \{2, ..., p-2\} \\ A &\equiv \alpha^a \text{ mod p} \end{aligned}$$

Bob generates his private key b and public key B in a similar fashion. Then Alice sends her private key A to Bob and Bob conversely sends his public key B to Alice. In a final step, both parties derive the shared secret k. To do so, Alice computes

$$k \equiv B^a \text{ mod p}$$

and Bob computes

$$k \equiv A^b \text{ mod p}$$

Therefore, by generating four values in total, from which two are kept secret, the two parties are capable of agreeing on a session key k that is impossible to derive by merely intercepting the public keys A and B. Why is it the case that both parties compute the identical key k?

The proof that both parties compute the same session key k can be seen easily, when basic algebraic rules are applied. Alice computes

$$B^a \equiv (\alpha^b)^a \equiv \alpha^{ab} \text{ mod p}$$

and Bob computes

$$A^b \equiv (\alpha^a)^b \equiv \alpha^{ab} \text{ mod p}$$

which is equivalent.

Therefore, both parties share the same key. The shared key can be used to establish a secure communication between Alice and Bob with a symmetric encryption scheme such as AES. [PP09]

p must be a prime with at least 1024 bit size and is generated with a probabilistic prime-finding algorithm. The integer α is a primitive element (generator) in the cyclic group. The private keys a and b should originate from a cryptographically strong random number generator, such that no attacker can guess them. The public keys are computed using the square-and-multiply algorithm and are usually precomputed. [PP09]

Abstract Algebraic Fundamentals In this section, the algebraic basics that are necessary to understand the discrete logarithm problem are introduced. This sections contents are strongly influenced by [PP09].

A *group* is a set of elements G with an operation $\circ$ that combines two elements of G. Groups have the following properties:

1. The group operation $\circ$ is closed. The result of a $\circ$ operation is again in G.
2. $\circ$ is associative. That means that $a \circ (b \circ c) = (a \circ b) \circ c$
3. There is an neutral/identity element $1 \in G$ such that $\forall a \in G, a \circ 1 = 1 \circ a$
4. $\forall a \in G$ there is an inverse element $a^{-1} \in G$ such that $a \circ a^{-1} = a^{-1} \circ a = 1$
5. The group is called abelian or commutative if additionally $\forall a, b \in G, a \circ b = b \circ a$

[PP09]

For example, $(Z, +)$ is a group. This can be confirmed by checking each of the above group properties.

For cryptographic purposes, groups with a finite number of elements are required. The group Z_n^* is considered, which consists of integers $0, 1, n-1$ for which $gcd(i, n) = 1$. The cardinality of Z_n^* is $|Z_n^*| = n$, therefore the group is finite. Z_n^* forms an abelian group under multiplication modulo n. The identity element is 1.

In relation to the DLP, so called *cyclic groups* are of interest. A cyclic group G is said to be cyclic if it contains an element α with maximum order $ord(\alpha) = |G|$. The order

$ord(a)$ of an element a of a group $(G, \circ)$ is the smallest positive integer k such that

$$a^k = a \circ a \circ ... \circ a = 1$$

Therefore, for G to be cyclic, there must be at least one element $\alpha \in G$ with $ord(\alpha) = |G|$. Such elements with maximum order are called *primitive* or *generators.* [PP09]

For example $|Z_{11}^*|$ has an primitive element $a = 2$, because $ord(2) = |Z_{11}^*| = 6$.

It's a fundamental theorem that for every prime p, $(Z_p^*, \cdot)$ is an abelian cyclic group. [PP09]

There are more properties to finite cyclic groups G.

1. The number of primitive elements of G is $\phi(|G|)$ where ϕ is Euler's totient function.
2. And if $|G|$ is prime, then all elements $a \neq 1 \in G$ are primitive.

[PP09]

Any element of a cyclic group is the generator of a subgroup, which in turn is also cyclic [PP09]. If $(G, \circ)$ is a cyclic group, then every element $a \in G$ with $ord(a) = s$ is the primitive element of a cyclic subgroup with s elements.

Lagrange's theorem states that if H is a subgroup of G, then $|H|$ divides $|G|$.

Furthermore, let G be a finite cyclic group of order n and let α be an generator of G. For each integer k which divides n, there exists exactly one cyclic subgroup H of G of order k. This subgroup is generated by $\alpha^{n/k}$. H consists exactly of the elements $a \in G$ which satisfy the condition $a^k = 1$. There are no other subgroups. This allows the construction of a subgroup from a given cyclic group, if the cardinality $|G|$ and a generator α of G is known. [PP09]

The Discrete Logarithm Problem In the previous sections, sufficient abstract algebraic fundamentals have been defined to introduce the DLP in Z_p^*.

Let Z_p^* be a finite cyclic group of order $p - 1$ and a *generator* $\alpha \in Z_p^*$ and another element $\beta \in Z_p^*$. The Discrete Logarithm Problem is the computational infeasibility of determining the integer $1 \leq x \leq p - 1$ such that:

$$\alpha^x \equiv \beta \text{ mod p}$$

There must exists such an integer x, since α is a generator and each group element can be expressed as a power of any primitive element.

x is the *discrete logarithm* of β to the base α:

$$x = log_\alpha \beta \text{ mod p}$$

Computing the discrete logarithm modulo a prime is a computationally hard problem, as long as the parameters are sufficiently large [PP09]. Exponentiation, $\alpha^x \equiv \beta$ mod p is the inverse function and can be performed in polynomial time (for example with the square and multiply algorithm).

The discrete logarithm is often used in groups with cardinality of a prime number in order to prevent certain attacks. Because $|Z_p^*| = p - 1$ has not prime cardinality, the discrete logarithm is used in subgroups of Z_p^*. [PP09]

Because this thesis researches the WPA3 Dragonfly handshake and most implementations merely support the discrete logarithm problem in elliptic curves (opposed to multiplicative groups Z_p^*), the **Generalized Discrete Logarithm Problem (GDLP)** is introduced.

Given a finite cyclic group G and a group operation $\circ$ and cardinality n. The GDLP is the computationally hard problem of finding the integer x, such that $1 \leq x \leq n$ and $\beta = \alpha^x$. Such an integer x must exist, because α is a primitive element and in one *cycle*, each element of the group is generated exactly once. There exists a range of different mathematical groups that have been used in practice for the GDLP.

1. Multiplicative group Z_p^* or a subgroup of it. This group was discussed in this brief introduction. DSA for example builds on top of it.
2. Cyclic groups from elliptic curves or hyperellipitic curves or algebraic varieties of it. Elliptic curves will be introduced in the next section.
3. The multiplicative group of a Galois field $GF(2^m)$ or a subgroup of it can be used to construct the DLP. $GF(2^m)$ will not be covered in this cryptographic introduction.

[PP09]

Attacks against the Discrete Logarithm Problem Since SAE makes use of a modified Diffie-Hellman Key Exchange, it is interesting to briefly list the existing (theoretical) attacks against the Discrete Logarithm Problem. The intractability of the DLP is based on the hardness of computing x for a given α and β in G such that $\beta = \alpha^x$.

Paar et al. state in their book (2009) that nobody really knows if there is an efficient algorithm to solve the GDLP, similar to the lack of knowledge of an efficient factorization method for the one-way function behind RSA [PP09].

There are **Generic Discrete Logarithm Algorithms** that make use of the group operation and no other algebraic structure of the group. They can be applied on any cyclic group. The running time of generic algorithms either depend on the size of the cyclic group or the size of the prime factors of the group order. [PP09]

The **Brute Force Search** is the easiest attack against the DLP. An attacker simply consecutively iterates over powers $1 \leq x \leq |G|$ and tests if $\alpha^x = \beta$. The complexity of this algorithm is thus $\mathcal{O}(|G|)$.

The **Baby-Step Giant-Step Algorithm** is a time-memory tradeoff algorithm. The basic idea is a divide and conquer approach, where $x = log_\alpha \beta$ is rewritten as $x = x_g m + x_b$ for $0 \leq x_g, x_b < m$ and m is chosen as $m = \sqrt{|G|}$. By rewriting the discrete logarithm, x_g and x_b can be searched separately. The baby-step giant-step algorithm requires computational runtime and memory capacity of $\mathcal{O}(\sqrt{|G|})$. [PP09]

The **Pollard's Rho Method** is a probabilistic algorithm with similar runtime as the baby-step giant-step method but almost no memory complexity [PP09]. The algorithm builds on top of the birthday paradox by pseudorandomly generating group elements of the form $\alpha^i \cdot \beta^j$ by walking randomly through the group. This process is repeated as long as no collision of two elements is obtained:

$$\alpha^{i_1} \cdot \beta^{j_1} = \alpha^{i_2} \cdot \beta^{j_2}$$

where $i_1 \neq i_2$ and $j_1 \neq j_2$ with $i_{1,2}, j_{1,2} \in \{1, p-1\}$

As soon as such a collision has been found, the discrete logarithm x can be easily computed. Pollard's rho algorithm is currently the best known attack to compute discrete logarithms in elliptic curve groups with an attack complexity of $\mathcal{O}(\sqrt{|G|})$. Therefore, the size of elliptic curve cryptosystems is required to be at least 2^{256}. [PP09; BSI19]

The **Pohlig-Hellman Algorithm** is based on the *Chinese Remainder Theorem* and exploits the factorization of the order of a group. This divide-and-conquer algorithm computes smaller discrete logarithms in the subgroups and then reconstructs x by using the Chinese Remainder Theorem. The individual subgroup discrete logarithms can be computed with Pollard's rho method. The runtime of Pholing-Hellman Algorithm depends on the prime factors of the group order. To use this attack, the prime factorization of the group order is required. [PP09]

One of the non-generic attack algorithms against the DLP is the **Index-Calculus algorithm** that specifically targets Z_p^* and $GF(2^m)^*$. The index-calculus method is very efficient with subexponential running time. For example, in order to obtain a security of 2^{80}, the primer number p of the DLP in Z_p^* should be at least 2048 bits. Due to the powerful nature of the Index-Calculus method, when using Z_p^* for the Diffie-Hellman Key Exchange, the prime p should have a minimal size of 2048 bits, or even 3072 bits to achieve better long term security. The German Federal Office for Information Security suggest at least 2048 bits until the year 2022, after that at least 3072 bits. [BSI19; PP09]

2.2 Elliptic Curve Cryptosystems

In this section Elliptic Curve Cryptography (ECC) is introduced, because WPA3-SAE mainly uses ECC in their implementations. `hostapd 2.8` for instance is configured to support only ECC out-of the box in the Dragonfly handshake. To enable multiplicative groups, the configuration must be explicitly changed.

Elliptic Curve Cryptography was invented in 1986-1987 by Neal Koblitz and Victor Miller. In the 1990s, it was not completely obvious that elliptic curves could be used to build asymmetric cryptosystems. However, after standardizations in banking standards in 1999 and 2001 and adoption in the IPsec protocol and TLS, they are nowadays considered to be secure [PP09].

ECC offers the same security guarantees as RSA or cryptosystems based on the DLP in Z_p^* and $GF(2^m)^*$, by using much shorter bitlengths. As of 2019, the bitlengths considered to be secure were at least 256 bits in ECC and at least 2048 bits in Z_p^* and $GF(2^m)^*$ [BSI19]. ECC is based on the generalized discrete logarithm problem, which means that elliptic curves need to have a cyclic group where the discrete logarithm problem is computationally intractable. [PP09]

Paar et al. define elliptic curves as follows:

An elliptic curve E over Z_p, $p > 3$ with p prime is the set of all pairs $(x, y) \in Z_p$ such that

$$E : y^2 \equiv x^3 + ax + b \bmod p$$

with an imaginary point at infinity $\mathscr{O}$ and $a, b \in Z_p$. Furthermore, the inequality $4a^3 + 27b^2 \neq 0$ mod p most hold. This inequality condition ensures that that curve plot has no self-intersections. [PP09]

The group operation in elliptic curves is **point addition** and **point doubling**. Both operations have a distinct meaning from a geometric point of view. In this quick introduction, the algebraic definition is of primary interest: [PP09]

$$P + Q = R \iff (x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

where

$$\begin{aligned} x_3 &= s^2 - x_1 - x_2 \text{ mod p} \\ y_3 &= s(x_1 - x_3) - y_1 \text{ mod p} \end{aligned}$$

and the slope s is defined as

$$\text{if P} \neq \text{Q (point addition) } s = \frac{y_2 - y_1}{x_2 - x_1} \text{ mod p}$$

$$\text{if P} = \text{Q (point doubling) } s = \frac{3x_1^2 + a}{2y_1} \text{ mod p}$$

The neutral element $\mathscr{O}$ is an abstract point at infinity such that

$$P + \mathscr{O} = P$$

Furthermore, an *inverse element* is required to fulfill the definition of a group G. The inverse of an group element P is $-P$, such that $P + (-P) = \mathscr{O}$. The inverse $-P$ is the point reflected on the x-axis:

$$-P = (x_p, p - y_p)$$

The points on an elliptic curve together with $\mathscr{O}$ have cyclic subgroups. When certain conditions are met, all points on a elliptic curve form a cyclic group [PP09]. No proof or further elaboration on the above statement is given, because the insight gained is of no importance in this thesis.

In order to make statements about the security of a elliptic curve, the order of the group is needed. *Hasse's* theorem gives a range to the number of points Z of an elliptic curve E modulo p:

$$p + 1 - 2\sqrt{p} \leq Z \leq p + 1 + 2\sqrt{p}$$

From the above statement it follows that the number of points is more or less in the range of the prime p [PP09].

The discrete logarithm for an elliptic curve E can now be defined as follows. Let P be an primitive element and T be another element of the curve. The discrete logarithm problem is finding the integer d, such that $1 \leq d \leq Z$ with Z being the number of points on E, such that:

$$P + P + ... + P = dP = T$$

[PP09] d is an integer and the private key in cryptosystems. For an attacker to break the elliptic curve cryptosystem, he has to figure how many times he needs to perform a point addition with P to reach the curve point T. The public key T is a point on the curve $T = (x_T, y_T)$. Multiplying the private key d with P is not directly possible, instead a repeated point addition is performed.

In order to perform point multiplication for elliptic curves, the Double-and-Add algorithm is used which is the counterpart of the Square-and-Multiply algorithm for multiplicative groups Z_p^*. This algorithm enables the computation of efficient point multiplications in elliptic curves. Now, sufficient concepts about elliptic curves and the discrete logarithm problem have been introduced in order to construct the Diffie-Hellman key exchange using elliptic curves (ECDH). [PP09]

The domain parameters for the elliptic curve Diffie-Hellman key exchange is a prime p and the elliptic curve E

$$E : y^2 \equiv x^3 + ax + b \bmod \text{p}$$

The primitive element $P = (x_p, y_p)$ is selected. The chosen curve E with its coefficients a, b and the prime p and the primitive element P are the domain parameters. Not every curve is suitable to build an computationally intractable discrete logarithm problem. However, the actual Diffie-Hellman Key Exchange is very similar to the DLP in the multiplicative group Z_p^*. [PP09]

First each party (Alice or Bob) choses a private key $a, b \in \{2, 3, Z - 1\}$ and computes their corresponding public key $A = aP$ and $B = bP$. Then Alice sends A to Bob and Bob in return sends B to Alice. Then both parties compute a joint secrete key T. Alice computes $T = aB = a(bP)$ and Bob computes $T = bA = b(aP)$ and since point addition is associative, both parties obtain the same key T. Paar et al. notes that in practical protocols, only the x-coordinate of the resulting key T is used. [PP09]

But why do real world implementations of WPA3-SAE such as `hostapd` use elliptic curves instead of other mathematical structures such as Z_p^*. The reason is that the most powerful algorithms for attacking the discrete logarithm problem in elliptic curves are not applicable, especially the **Index-Calculus** method. Merely the **Baby-Step-Giant-Step** algorithm and **Pollard's Rho** remain, which both reduce the attack complexity to $\sqrt{|G|}$. In practice, elliptic curves with group order of at least 2^{256} should be used [BSI19]. It is not easy to find a cryptographically strong elliptic curve with prime order, therefore they are often standardized by organizations such as the National Institute of Standards and Technology (NIST) [PP09].

3 The SAE Handshake

3.1 SAE is a PAKE Scheme

The Simultaneous Authentication of Equals (SAE) handshake is essentially a Password Authenticated Key Exchange (PAKE) scheme. PAKE addresses a practical security problem: How can two parties with a shared secret establish secure, authenticated communication without relying on a Public Key Infrastructure (PKI)? [HR10]

PAKE schemes are authenticated key exchanges. Two parties have a shared secret and deterministically establish a high entropy password using zero-knowledge proofs of the authenticated password. PAKE schemes build on top of public key cryptography which consists of mathematical building blocks that provide a cryptographic one-way function that is hard to compute in one direction, but is efficiently computed in the opposite direction. One of the first public key exchange mechanism invented was the **unauthenticated** Diffie-Hellman key exchange. However, the Diffie-Hellman key exchange is vulnerable against man-in-the-middle attacks.

A man-in-the-middle attacker can completely break the security of the standard Diffie-Hellman key exchange by intercepting the public values of the involved participants and substituting them with the attackers public key value. Both parties thus agree to a shared secret with the attacker. The man-in-the-middle attacker then simply decrypts all messages exchanged between the two parties and reads and modifies them at will and re-encrypts the messages with the previously negotiated key, before transmitting it back to the other party. This attack is possible because the key exchange is not authenticated. [PP09]

PAKE schemes on the other side provide mutual authentication through an common secret that was established in a out-of-band mechanism. An eavesdropping attacker cannot obtain any information about the password while observing the exchange. She is merely capable of making an online guess at the password during the handshake execution. [HR10]

PAKE schemes make use of zero-knowledge proofs. A zero-knowledge proof (ZKP) is a method to authenticate two parties without exchanging the actual passwords, therefore not creating a possibility of password disclosure. A ZKP allows a participant to prove knowledge over a secret without actually revealing it. [HR10]

Many PAKE techniques such as EKE (Patent [BM19]) and SPEKE (Patent [Jab19]) are patented and could not be used in the development of new PAKE schemes such as Dragonfly. Nowadays, those patents have long been expired as of 2019 [HR10]. Legal issues originating from patents were the main motivation for Dragonfly's designers to obfuscate the immanent Diffie-Hellman key exchange included in PAKE schemes with a random mask [Per19]. Hao et al. note that patenting issues are *"arguably one of the biggest brakes in deploying PAKE solutions in practice"* [HR10].

The security properties that a PAKE protocol provides are the following:

1. **Offline dictionary attack resistance** - A passive or active attacker cannot collect sufficient key exchange material in order to launch an offline brute force attack against the password.
2. **Perfect forward secrecy** - The negotiated keys are secure in the face of a future disclosure of passwords.
3. **Known session security** - A disclosed session does not affect the security of other established session keys.
4. **Online dictionary attack resistance** - Only one password guess per protocol execution is possible.

[HR10]

Schemes holding the above properties are known as balanced PAKE schemes.

Hao et al. note that there is an extra security requirement called *server compromise resistance* that makes a PAKE scheme an augmented PAKE scheme if *"an attacker should not be able to impersonate users to a server after he has stolen the password verification files stored on that server, but has not performed dictionary attacks to recover the passwords"* [HR10]. Furthermore, Hao et al. question the necessity of the resistance against server compromise by arguing that the all passwords would need to be revoked in the case of server takeover in either case. Additionally, they note that a balanced PAKE could hash and salt the passwords which renders any advantages from the *server compromise resistance* redundant [HR10].

3.2 Dragonfly

The **Simultaneous Authentication of Equals (SAE)** handshake was proposed by Daniel Harkins in 2008 [Har08]. The IEEE 802.11s amendment for wireless mesh networks made first use of the SAE handshake [VR19]. The name **Dragonfly** refers to the architectural identical handshake defined in RFC 7664 by Daniel Harkins [Har15]. Clarke et al. also notice the similarity of the two handshakes in their cryptanalysis of Dragonfly [CH14].

The terms SAE and Dragonfly are used somewhat interchangeably during this thesis. The IEEE 802.11 standard of 2016 defines SAE as a variant of the Dragonfly family [IEE16]. Maty Vanhoef et al. note that there exist several minor variants of the SAE handshake and the family of those handshake is referred to as Dragonfly. Dan Harkins also proposed a TLS-PWD authenticated key exchange named *Secure Password Ciphersuites for Transport Layer Security (TLS)* in RFC 8492 that can be ascribed to the Dragonfly family [Har19a].

Throughout this thesis, the names **SAE**, **Dragonfly**, **WPA3-SAE** or **WPA3-Personal** refer to the same password authenticated key exchange defined in RFC 7664 as a candidate for general Internet use [Har15]. The Dragonfly handshake is certified by the Internet Engineering Task Force as WPA3-Personal and standardized in the IEEE Standard 802.11 Std 2016 [IEE16].

SAE guarantees the following security properties:

1. After successful completion of the SAE handshake, the handshake participants share a high entropy pairwise master key (PMK)
2. No attacker who passively or actively intercepts and manipulates the handshake may obtain the password or the resulting PMK
3. Only one guess at the password per execution of the handshake is possible. Therefore, no offline dictionary attacks are feasible
4. Knowing the PMK from a previous execution of the handshake doesn't give an attacker an advantage in future executions
5. The revealing of the password cannot be leveraged to decrypt past traffic or reconstructing past PMK's

[IEE16]

Dragonfly is a symmetric peer-to-peer protocol that allows two parties to bootstrap a secure symmetric key from a low-entropy shared secret over insecure public channels [CH14]. The two participants are mutually authenticated based on whether they have identical passwords [CH14].

Both sides can initiate the handshake simultaneously. Dragonfly can be used in a traditional client-server architecture, but also in peer-to-peer applications where either side may initiate the handshake [Har15]. Therefore, there are no clear supplicant and authenticator roles, the participating parties are *equals*.

Dragonfly is based on discrete logarithm cryptography build on elliptic curves (ECC) or finite fields (FFC) using multiplicative groups module a prime p. There are two message exchanges in the Dragonfly protocol: The **commit** and **confirm** exchange.

A party can commit at any time. But only after both parties proceeded with the commit exchange, the confirm exchange is possible. The purpose of the commit exchange is to force each participant to a single guess at the password [IEE16]. The confirm message exchange on the other hand is used to prove the correctness of this prior password guess. Authentication is accepted as soon as a peer confirms and the protocol successfully terminates after both parties confirmed [Har15]. The commit step can happen at any time, confirmation however is only allowed after both participants committed. A participant accepts the authentication after a peer has successfully confirmed. When both peers accept, the handshake successfully terminates. [IEE16]

A full state machine of the handshake is introduced in section 5.4. A high level overview of the cryptographic operations in the SAE handshake is given in figure 1.

The 802.11 standard requires SAE implementations to support the elliptic curve NIST P-256 as a minimum [VR19]. Vanhoef et al. remark that support for other groups is optional and that there exists no requirement to support multiplicative groups [VR19]. In the remainder of this chapter, only Dragonfly for elliptic curve cryptography is covered, because all relevant open source implementations prefer SAE with ECC [Jou19; Int19].

In the Dragonfly handshake, all elliptic curves are defined over the equation

$$y^2 = x^3 + ax + b \text{ mod p}$$

where p is a prime that determines a prime field $GF(p)$. The cryptographic group is a subgroup of the full elliptic curve group that consists of the elliptic curve's points with the point at infinity O, which is the identity element of the group [Har15]. The elements $a, b \in GF(p)$ define the curve's equation. The elliptic curve has a point G, that serves as a generator of the group with sufficiently large order $q = |G|$. The order q is prime and should generate a sufficiently large subgroup that is considered secure [Har15].

The standard mentions that only ECC groups defined over an odd prime finite field with a cofactor of 1 should be used [IEE16]. The element operation is defined as addition of two points on the curve, resulting in a third point on the curve. The scalar operation is the multiplication of a point on the curve by a scalar, resulting in a new point on the curve. The inverse operation is inversion of a point on the curve, resulting in a second point on the curve. [IEE16]

RFC 7664 also lists some assumptions about the Dragonfly protocol. The most important ones being that the Dragonfly peers should be able to produce cryptographically strong random numbers and that the discrete logarithm problem for the chosen group must be hard [Har15]. This means that given an ECC group E with an generator G, and the result of the scalar operation $Y = x \cdot G$, it is computationally infeasible to determine x [Har15].

In the remainder of the chapter, the two participants of the protocol are referred to as supplicant (station) and authenticator (access point). Even though the Dragonfly handshake may be initiated simultaneously by both participants, the typical supplicant/authenticator configuration is of interest during this thesis. An example for such an configuration is an Android smartphone trying to establish a connection to a router via WPA3-Personal. In this case, the Android phone (supplicant) initiates the Dragonfly handshake by sending an Auth-Commit frame. The authenticator (home router) replies with an Auth-Commit and Auth-Confirm frame. The Android smartphone finalizes the handshake by sending an Auth-Confirm frame.

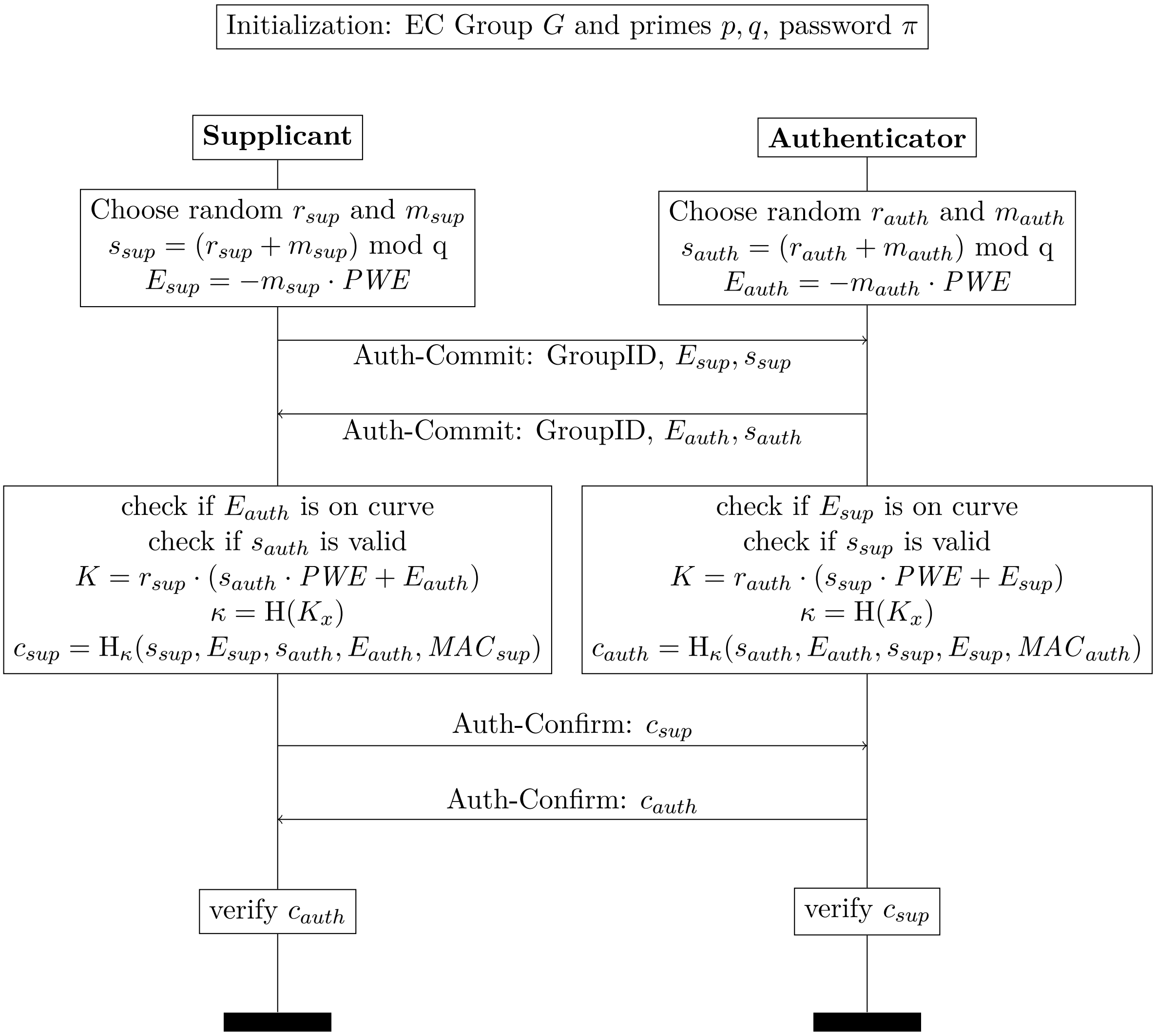
Initialization: EC Group G and primes p, q, password π
Supplicant
Authenticator
Choose random r_{sup} and m_{sup}
$s_{sup} = (r_{sup} + m_{sup})$ mod q
$E_{sup} = -m_{sup} \cdot PWE$
Choose random r_{auth} and m_{auth}
$s_{auth} = (r_{auth} + m_{auth})$ mod q
$E_{auth} = -m_{auth} \cdot PWE$
Auth-Commit: GroupID, E_{sup}, s_{sup}
Auth-Commit: GroupID, E_{auth}, s_{auth}
check if E_{auth} is on curve
check if s_{auth} is valid
$K = r_{sup} \cdot (s_{auth} \cdot PWE + E_{auth})$
$\kappa = \mathrm{H}(K_x)$
$c_{sup} = \mathrm{H}_\kappa(s_{sup}, E_{sup}, s_{auth}, E_{auth}, MAC_{sup})$
check if E_{sup} is on curve
check if s_{sup} is valid
$K = r_{auth} \cdot (s_{sup} \cdot PWE + E_{sup})$
$\kappa = \mathrm{H}(K_x)$
$c_{auth} = \mathrm{H}_\kappa(s_{auth}, E_{auth}, s_{sup}, E_{sup}, MAC_{auth})$
Auth-Confirm: c_{sup}
Auth-Confirm: c_{auth}
verify c_{auth}
verify c_{sup}

Figure 1: High-level overview of the SAE commit and confirm handshake [LŠ15]. In this figure, the variable names *scalar*, *Element* and *mask* are abbreviated as s, E and m. H_κ is a cryptographic hash function with key κ and is usually implemented as SHA256-HMAC. K_x is the x-coordinate of the elliptic curve point K.

3.3 Deriving the Password Element

In the beginning of the Dragonfly key exchange, the shared password π is chosen at random and given to the supplicant and authenticator in a secure out-of-bounds process (for example shouting from door to door in a office space). Then both parties compute a base password element PWE using a *hunting-and-pecking technique* that repeatedly creates a seed

$$\mathit{SHA256}(MAC_{sup}, MAC_{auth}, \pi, c)$$

with a counter c that is incremented each round in a loop. The resulting hash digest is translated into a potential x-coordinate of an elliptic curve point with a standardized key derivation function. [Har15]

The key derivation function KDF-n is defined as a function that takes a key k to stretch, a label to bind to the key and a desired output length n [Har15]. In RFC 7664, the KDF is not defined. In `hostapd 2.8`, the KDF-n is a SHA256-HMAC which uses the seed as key [Har15].

The x-coordinate is plugged into the curve equation ($y^2 = x^3 + ax + b$ mod p) and it is confirmed whether the result of the equation is a quadratic residue modulo p. If this holds, the password element PWE has been found, otherwise the counter is incremented and a new seed is generated [Har15].

This quadratic residue modulo p test is done with the Legendre function [VR19]. To prevent timing attacks, all possible loop iterations ($k = 40$) are always performed, even if a valid point was found earlier.

The check whether a value is a quadratic residue modulo a prime can leak information about the value in a side-channel attack [Har15]. For this reason, the residuosity check is blinded with a random value. The core idea is that all cryptographic functions need to be computed in constant time, independent from a specific cryptographic outcome [Har15; VR19]. Otherwise, timing attacks would be eligible.

3.4 Commit Exchange

In the commit exchange, both parties exchange scalars and elements and generate a shared secret K as a result. They *commit* to a single guess of the password [Har15].

The supplicant generates two random numbers

$$\begin{aligned} 1 < r_{sup} < q \\ 1 < mask_{sup} < q \end{aligned}$$

and the authenticator generates respectively

$$\begin{aligned} 1 < r_{auth} < q \\ 1 < mask_{auth} < q \end{aligned}$$

The sum of r and $mask$ must also lie in the same range $1 < scalar_{sup|auth} < q$.

Both participants compute the public scalars and elements based on their private random numbers. After this step, the mask is no longer needed and must be irretrievably destroyed [Har15].

$$\begin{aligned} scalar_{sup} &= (r_{sup} + mask_{sup}) \mod q \\ Element_{sup} &= -mask_{sup} \cdot PWE \end{aligned}$$

and the authenticator computes equivalently

$$\begin{aligned} scalar_{auth} &= (r_{auth} + mask_{auth}) \mod q \\ Element_{auth} &= -mask_{auth} \cdot PWE \end{aligned}$$

Figure 1 illustrates a high level overview of the SAE handshake. In this figure, the variable names $scalar$, $Element$ and $mask$ are abbreviated as s, E and m. H refers to the hash function in use, which is HMAC-SHA256.

After both participants exchange their scalars and elements in a Auth-Commit authentication frame, they both verify the correctness of the scalars ($2 < scalar < q$) and elements. A valid element must consist of coordinates that are non-negative integers and less than the prime p [IEE16]. The element cannot be the point at infinity and should be a valid point on the curve [IEE16]. If the peer has sent an identical scalar and element, it is considered to be a reflection attack and the handshake must be aborted [Har15].

Otherwise the participants compute the shared secret K. The supplicant computes

$$K = r_{sup} \cdot (scalar_{auth} \cdot PWE + Element_{auth})$$

and the authenticator computes

$$K = r_{auth} \cdot (scalar_{sup} \cdot PWE + Element_{sup})$$

If the shared secret K is the point-at-infinity, the participants shall reject the authentication [IEE16]. The x-coordinate of the shared key K is fed into a hash function to derive the key κ:

$$\kappa = Hash(K_x)$$

Usually κ is stretched into two subkeys for cryptographic hygiene. One subkey is used as a key confirmation key (KCK) for the computation of the confirm token, the other subkey (PMK) is used as input to the subsequent 4-way handshake. Each subkey must be at least the length of the prime used in the selected group, enforcing the key derivation function to have a output of at least $n = len(p) \cdot 2$ which is usually 512 bits [Har15]. Both subkeys thus have a size of 256 bits, which is sufficient considering the suggestions about key lengths from the Federal Office for Information Security of Germany (As of 2019) [BSI19; IEE16].

3.5 Confirm Exchange

In the confirm exchange, both parties verify that they derived the same secret κ and therefore possess the same password π. The confirm token consists of a HMAC-SHA256 digest created with the key κ with the handshake summary as input.

The supplicant computes:

$$c_{sup} = \text{HMAC-SHA256}_{\kappa}(scalar_{sup}, Element_{sup}, scalar_{auth}, Element_{auth}, MAC_{sup})$$

and the authenticator computes accordingly her confirm token

$$c_{auth} = \text{HMAC-SHA256}_{\kappa}(scalar_{auth}, Element_{auth}, scalar_{sup}, Element_{sup}, MAC_{auth})$$

Then both the supplicant and authenticator exchange their tokens and verify the other tokens. This verification is possible because all necessary elements were exchanged previously and are known to all participants. If the verification succeeds, the handshake completes and the subkey κ is used as a PMK. Otherwise, the handshake times out and the authentication fails.

3.6 Security

The core security idea underlying the Dragonfly handshake is that the adversarial advantage grows through interaction and not through computation. No information other than the knowledge whether a single guess at the password was correct, is leaked.

The security of the scheme is based on the computational Diffie-Hellman assumption (CDH) which states the computational intractability of computing the value g^{ab} given (g, g^a, g^b) where g is a randomly chosen generator of the cyclic group G of order q and $a, b \in \{0, ..., q-1\}$. If the computation of the discrete logarithm to the base g was computationally feasible, then the CDH problem could be solved by computing a by taking the discrete logarithm of g^a and then deriving g^{ab} by exponentiation of $(g^b)^a$. [Har19b; LŠ15]

This computational Diffie-Hellman assumption exists analogously in Dragonfly. It's computationally intractable to compute the value $\mathrm{PWE}^{r_{sup} \cdot r_{auth}}$ given

$$(r_{sup} + mask_{sup}, \mathrm{PWE}^{-mask_{sup}}, r_{auth} + mask_{auth}, \mathrm{PWE}^{-mask_{auth}}, \mathrm{PWE})$$

where PWE is a generator of the cyclic group G of order q,
$r_{sup}, r_{auth}, mask_{sup}, mask_{auth} \in \{1, ..., q-1\}$ and the sum $r + mask \in \{1, ..., q-1\}$. If the computation of the discrete logarithm to the base PWE was possible, then the security of the Dragonfly protocol could be broken by computing the discrete logarithm of $\mathrm{PWE}^{-mask_{sup|auth}}$ thus obtaining $-mask_{sup|auth}$, taking its inverse and finding $r_{sup|auth}$ by subtraction:

$$r_{sup|auth} = scalar_{sup|auth} - mask_{sup|auth}$$

which enables the computation of the value $\mathrm{PWE}^{r_{sup} \cdot r_{auth}}$ by exponentiation. [Har19b]

Security Proof of Dragonfly A major point of criticism of Dragonfly was the lack of a security proof for the handshake. This changed when Lancrenon et al. published their paper *On the Provable Security of the Dragonfly Protocol* in 2015 which proves a close variant of Dragonfly to be secure in the random oracle model [LŠ15].

The paper proves that the core mathematical structure of Dragonfly - a Diffie-Hellman variant with a password-derived base - is secure in the random oracle model [LŠ15]. The security of Dragonfly is based on the Computational Diffie-Hellamn (CDH) and Decisional Inverted-Additive Diffie-Hellman (DIDH) assumptions. The authors orient their work on existing proofs of older PAKE protocols.

For the objectives of this thesis, the internals of a cryptographic proof are of secondary nature, because the main goal is to find programming flaws by fuzzing frames and not breaking the underlying cryptographic scheme.

3.7 Practical Attacks against SAE

RFC 7664 mentions that salting of passwords is redundant, because salted password are merely new passwords used for authentication. If an attacker is capable of obtaining the salted password, she can authenticate herself to the other participant [Har15].

RFC 7664 also suggests to limit the number of online guesses at the password by refusing authentication after a certain number of failed attempts in order to prevent brute force attacks [Har15]. Furthermore, the RFC recommends to use a minimal number of $k = 40$ iterations to reduce the probability that no suitable password seed was found to roughly one in a trillion [Har15].

Vanhoef et al. presented a variety of attacks in the paper *Dragonblood: A Security Analysis of WPA3's SAE Handshake* against practical Dragonfly implementations [VR19].

Dragonfly used to be susceptible to side-channel leaks in its password derivation algorithm. The first implementation stopped the loop execution as soon as a valid quadratic residue modulo p was found. This vulnerability was fixed in 2011, when 40 iterations were introduced regardless of whether a valid point was found in an earlier iteration [VR19].

Another fix was the introduction of quadratic residue blinding that prevents leaking information of the Legendre Function, which basically implements the *Double-And-Add Algorithm*. Vanhoef et al. wondered why the MAC address was added to the seed in the password derivation algorithm. They authors explain that the algorithm doesn't need to be executed before every handshake execution and that without MAC addresses as inputs, the password element could be computed offline. This would also reduce the susceptibility to DoS attacks [VR19].

Targeting SAE's Anti-Clogging mechanism The SAE handshake added defenses against side-channel leaks that are caused by the password derivation function. Vanhoef et al. demonstrate how those defensive mechanisms can be abused in a DoS attack. [VR19]

The anti-clogging mechanism is necessary because access points perform computational expensive operations when receiving a SAE commit frame. This can be abused by flooding an access point with spoofed commit frames with random MAC addresses. The defenses against side-channel leaks further increase the computational costs for such a commit frame, because each password derivation execution requires constant amount of work. [VR19]

The anti-clogging mechanism consists of a reflection of a cookie that the authenticator generates for each supplicant. The cookie itself is a hash over the connection id, initiator id and a random sequence of bytes generated by the access point. Those tokens are concatenated and hashed with SHA256 and must be reflected by the client in order to continue with the computationally expensive processing of the commit frame. This procedure prevents the client from creating new connections with spoofed connection ids. In the case of the Dragonfly handshake, the connection id is simply the MAC address. It is trivial to spoof MAC addresses, because the attacker can capture all anti-clogging tokens that are broadcasted in a 802.11 network and reflect the secret cookies. After the attacker acknowledged the anti-clogging frames, the access point agrees to process the commit frame. [VR19]

Vanhoef et al. targeted a non-disclosed professional access point and was capable of completely exhausting the resources of the CPU by sending 70 commit exchanges per second. By using the elliptic curve P-521, the impact was even more catastrophic [VR19].

Anti-clogging tokens have a timeout. In order to prevent those tokens from timing out, access point implementations either renew the anti-clogging tokens after a certain threshold or after a timeout.

According to Vanhoef et al., a possible countermeasure is the costly derivation of the password element in a low-priority background thread. This ensures that network functionality is not impacted, even though legitimate clients won't be able to connect to it. Vanhoef et al. suggest that the reason for the DoS attack is the online computation of the password element. If it would be possible to compute the token offline, the password element could be derived only once at startup of the station. [VR19]

Downgrade and Dictionary attack against WPA3 in transition mode Vanhoef et al. present a downgrade attack against WPA3-SAE transition mode. When an man-in-the-middle attacker tries to downgrade a client to use WPA2 instead of WPA3, even if the access point is capable of using WPA3, the downgrade attempt will be recognized in message 3 of WPA2's 4-way handshake. The attacker only needs a single authenticated 4-way handshake message to carry out a offline dictionary attack. It isn't necessary to use an man-in-the-middle attack, a rouge network may be alternatively used. In order to attack clients that use WPA3 in transition mode, many clients tried to autoconnect to the rouge WPA2 network. Because message 2 of the 4-way handshake is already authenticated, a offline dictionary attack can be launched. [VR19]

When sending a commit frame to an access point, the client has to choose a elliptic curve or multiplicative group. If the access point doesn't support the group, he will reply with the appropriate status code. This process continues until both participants agree on a group. Now an attacker can jam or forge channel-switch announcements in order to indicate to the access point that the client doesn't support this group. By doing so, a less secure group can be used. The reverse process is also possible, which might be interesting for Denial of Service attacks [VR19].

Timing attacks against WPA3's SAE groups Although elliptic curves are the default cryptographic structure for the Dragonfly handshake, multiplicative groups Z_p^* are also supported. Vanhoef et al. demonstrate that there are no side-channel defenses against Dragonfly with multiplicative groups. The general idea of this specific attack can be summarized as follows: Vanhoef et al. modified a tool on top of `aircrack-ng` to spoof commit frames and measure the time it takes to receive the corresponding commit reply. They proceeded by sending a deauthentication packet to the access point in order to clear the state related to the spoofed MAC address. This process was repeated hundreds of times and the average response times for different MAC addresses and different multiplicative groups were collected. Based on timing differences, they concluded the number of iterations the password derivation algorithm must have used. This attack was possible, because there were no side-channel defenses against multiplicative groups in the Dragonfly handshake [VR19].

Cache-based side channel attacks against SAE Vanhoef et al. tried to figure out if the quadratic residue test in the password derivation algorithm succeeded in the first iteration. The elliptic curve version of `hostapd` and `wpa_supplicant` has defenses against timing attacks in the form of constant time computations that do not leak any information about internal states of cryptographic computations. However, the password derivation algorithm might still be vulnerable to a micro-architectural side-channel attack. Those attacks abuse capabilities of modern processors to optimize their behavior based on past computations. [VR19]

With the **Flush+Reload** attack, an attacker flushes a memory location from the cache and waits for a specific time interval before reloading the flushed location. When the victim accesses the memory location, it will be cached and the reload time will be much shorter. By repeating those steps, an attacker can trace the victims memory locations [VR19].

Vanhoef et al. were able recognize with high certainty that the password element was found in the first iteration using cache-based side channel attacks. They monitored the access to the quadratic residue test by finding a branch in the iteration loop that resulted in two separate cache lines. With the help of the cache line that was executed in every iteration of the loop, they managed to create a synchronization clock. By using a linear classifier, Vanhoef et al. were able to make a probabilistic statement whether the password was found in the first iteration. [VR19]

Offline password partitioning attack Vanhoef et al. managed to perform password partitioning attacks with information learned in cache-based side channel attacks. They created a dictionary and tried to recover the password from it. They repeatedly partitioned the dictionary into correct and incorrect password candidates. When the dictionary became empty, the password could not have been in it. If only one password remained after the partitioning steps, it was with high probability the correct password. Vanhoefs algorithm used a set of element tests and their result and the MAC address of the target as input to partition the dictionary by removing passwords that lead to a different result for the element test compared to the result that was obtained by the timing or cache-based attack. The attack could be launched offline. [VR19]

On average, Vanhoef et al. needed to perform 24.3 timing measurements to obtain 35 element tests for the MODP group 22. For the elliptic curve P-256, an adversary required 29 element test results to uniquely recover the password with a probability above 95%. In order to launch an offline brute force attack against all possible 8-character lowercase passwords, 38.38 element tests for multiplicative group 22 and 38.92 for elliptic curve P-256 were required. In order to launch an successful offline dictionary attack on Amazon EC2 instances, an attacker needed to spend on average \$125 dollars for elliptic curves and \$10 dollars for multiplicative groups [VR19].

Conclusion Vanhoef et al. concluded that a simple change to the password encoding mechanism could have prevented almost all attacks. By excluding the peer's MAC address from the password encoding algorithm, it would be possible to compute the password element offline such that an attacker can no longer trigger costly executions of the password encoding mechanism. Furthermore, the execution time of the password encoding method would have been identical, such that no side-channel information can leak. [VR19]

4 Fuzzing Environment

The 802.11 Wi-Fi ecosystem is a collection of complex standards and certifications [IEE16]. Due to the unreliable and error prone nature of wireless protocols over radio frequencies, one first milestone of this thesis will be a practical guide on how to set up a laboratory environment where fuzzing tests against the WPA3-SAE handshake can be tested. As it turned out, finding a suitable 802.11 environment is not a trivial task. This chapter gives readers a practical guide, such that they are capable to reproduce the obtained results.

The operating systems used were `Ubuntu 18.04` (with kernel `4.15.0-39-generic`) and the `Kali Linux release 2019.1/2019.2` distribution.

Besides the built in wireless network card of the working machine used throughout this thesis (Wireless card `Intel Centrino Advanced-N 6235` with mac80211 driver `iwlwifi`), the following Wi-Fi cards have been used:

1. Panda Wireless Stick, Model: PAU07 with driver `rt2800usb`

2. AWUS036ACH Wireless Stick with driver `rtl8812au v5.3.4`

The central goal of this thesis is the fuzzing of authentication management frames in the WPA3 handshake. As figure 8 illustrates, the whole WPA3-Personal connection establishment includes beacon frames, probe request/response frames, authentication request and response frames, association request and response and the finalizing EAPOL 4-way handshake. Only authentication frames are targeted in the fuzzing framework.

The laboratory setup plays a major role in determining the effects of the fuzzing tests. The most important questions related to the testing environment are the following:

1. How costly is the setup of the fuzzing environment? Is additional hardware needed? Is the management of multiple operating systems in virtual environments necessary? Is the entire environment easily programmatically controllable?

2. How easy is it to measure the effects of fuzzing tests? Is it possible to see which fuzzed frame caused a crash in the targeted device?

3. Are software internals of the fuzzing target known? If no, a blackbox fuzzing approach is used, otherwise, a greybox strategy is applied.

First the different Linux 802.11 kernel subsystems are introduced to provide an overview of the Wi-Fi kernel architecture. Then various fuzzing architectures are outlined in the following sections and the chapter is concluded by choosing a suitable laboratory setup for this thesis.

4.1 Kernel 802.11 Architecture

The Linux Kernel 802.11 subsystem consists of three layers. The user-space layer with Wi-Fi applications such as `iw`, `wpa_supplicant` and `hostapd` forms the first layer. Then the interface between user-space and kernel is the `nl80211` netlink-based protocol that specifies how wireless devices are configured and managed. The kernel side of configuration management for wireless devices is `cfg80211`. Then there is the `mac80211` driver API that describes the interface for SoftMAC wireless cards.

A distinction can be made between **SoftMAC** and **FullMAC** wireless cards. In FullMAC wireless cards, the **Media Access Control (MAC) Sublayer Management Entity (MLME)** is managed completely in the NIC's hardware, whereas the MLME in SoftMAC wireless drivers resides on the main CPU and is implemented in `net/mac80211/mlme.c` within the Linux kernel source tree. The MLME is the management entity where the physical layer (PHY) of the MAC state machine is implemented. Therefore, the MLME is responsible for reaching relevant states in authentication/deauthentication and association/deassociation and beacon frames[7]. Put differently, in FullMAC Wi-Fi chips, the MLME including the MAC and PHY layers are fully placed in the firmware, whereas in SoftMAC devices, the MLME resides on the main CPU of the host machine. [wir19]

If the MLME is placed on the chip, the host processor may offload 802.11 functionality on the NIC processor in order to save memory and power. For this reason, mobile phones and IoT devices follow usually have a FullMAC architecture [Kav19]. Broadcom chips are FullMAC wireless drivers, such as the `brcm` firmware implemented in mobile phones. An example for `mac80211` driver usage are laptops with Intel Wi-Fi chips that are powered by the SoftMAC `iwlwifi` driver. Illustration 2 gives a visual overview of the Linux 802.11 architecture with examples for each type of 802.11 driver.

Usually, WPA3-SAE authentication is handled in user-space software such as `wpa_supplicant`. However, the Linux kernel supports SAE authentication offloading, such that the SAE handshake may be implemented in 802.11 chipsets (Compare section 6.3.2).

4.2 Using Virtualization and Emulation Software

The wireless supplicant implementation of Intel named `iwd` incorporates a testing framework that runs within QEMU [Bel19]. QEMU is software that allows to emulate processors and complete hardware of computers. Processor instructions of the guest processor are translated into CPU instructions of the host processor.

There are other virtualization solutions such as Oracles VirtualBox or VMware Workstation. Such virtualization software makes it possible to launch complete operating systems in a virtual environment and assign a Wi-Fi device directly to the virtual

[7] The MLME implementation of SoftMAC drivers within the `mac80211`, accessed on 30th July 2019, `https://github.com/torvalds/linux/blob/master/net/mac80211/mlme.c`

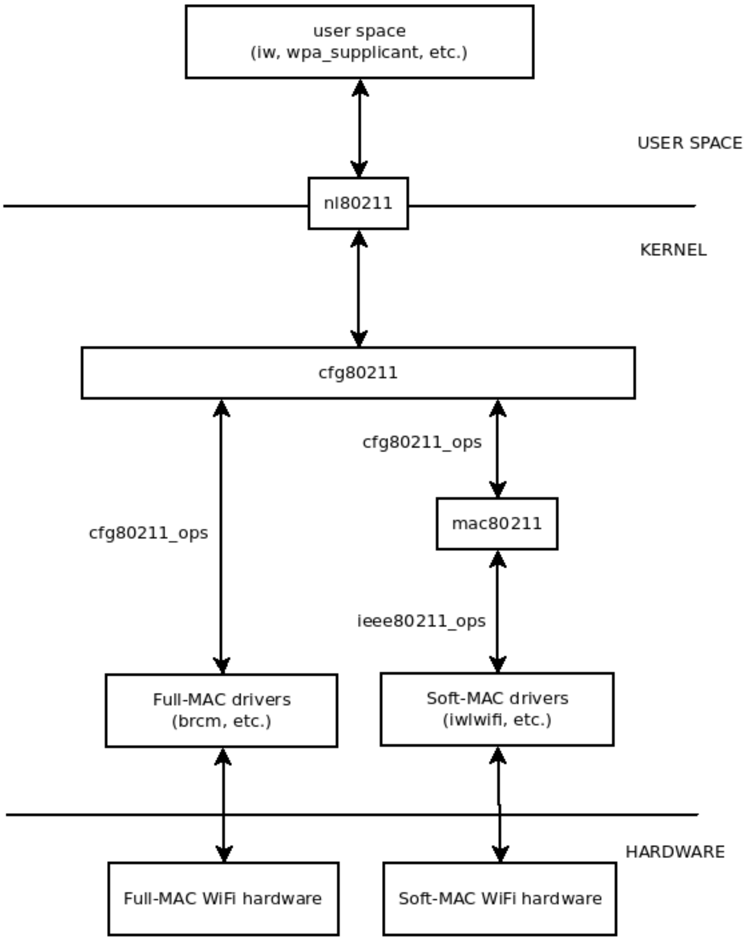

Figure 2: MAC subsystem of modern Linux kernels. [htt19]

machine. With such a setup, it is possible to run software such as `hostapd` in the virtual machine with a dedicated Wi-Fi device assigned and treat it logically as access point. An additional 802.11 device on the host machine is used as a fuzzing supplicant which targets the access point. The reverse scenario is also possible, where the AP is on the host machine and the fuzzing supplicant is created on the virtual machine.

The advantages are obvious: Virtual machines can be halted and freezed. Instructions and memory can be inspected after crashes and logging can be enabled. In simulators, time is discrete and may be stopped. Such a setup allows to keep full control over the targeted access point without the risk of losing any information after a crash caused by a fuzzing engine.

Using virtual machines allows programmatic control over the whole testing environment, even though additional 802.11 hardware is used. The virtual machine is usually on the same machine as the testing environment. The costs of the setup are moderate. However, the cognitive overhead when switching between host and virtual machine cannot be neglected.

Example: WPA3-SAE Access Point on Kali Linux `VMWare Player 15.0.4` and the `Kali Linux 2019.1 VMWare Image` will be used as operating systems. Then `hostapd 2.8` is compiled with SAE support on the Kali Linux operating system. In order to use VMWare and a OS image, secure boot needs to be disabled on the host computer.

On the host computer, `wpa_supplicant 2.8` will be setup and compiled with SAE support. Then a Wi-Fi network with key management SAE is launched by connecting the supplicant with the authenticator. If the connection is successful, a basic WPA3 network is obtained, even if protected management frames are not supported. In such a setup, both the access point and the supplicant are under the control of the fuzzer, which allows to write fuzzing tests targeting the WPA3-SAE handshake comfortably.

Instructions in the Kali Linux 2019.1 virtual machine Download and unpack `hostapd 2.8` with the instructions:

```
wget https://w1.fi/releases/hostapd-2.8.tar.gz
tar xzvf hostapd-2.8.tar.gz
cd hostapd-2.8/hostapd
```

After downloading the sources, several packages/libraries that are necessary to compile `hostapd` and the supplicant are installed.

```
apt install pkg-config
apt install libnl-3-dev
apt install libssl-dev
apt install libnl-genl-3-dev
```

Then the line `CONFIG_SAE=y` is appended at the end of the `defconfig` file located in `hostapd-2.8/hostapd`.

Now `hostapd-2.8/hostapd` can be compiled with the following instructions:

```
cp defconfig .config
make -j 2
cd ..
```

The compilation yields a fresh `hostapd` binary with SAE key management support. The next step is to configure `hostapd-2.8/hostapd` to use WPA3-SAE.

A configuration file named `wpa3.conf` is created with the following contents:

```
interface=wlan0
ssid=WPA3-Network
hw_mode=g
channel=1
wpa=2
wpa_passphrase=abcdefgh
wpa_key_mgmt=SAE
rsn_pairwise=CCMP
#ieee80211w=2
```

The crucial line `wpa_key_mgmt=SAE` instructs `hostapd` to use SAE as key management protocol. Before starting `hostapd`, it is necessary to kill all programs that might interfere with it. A bash script named `prepare.sh` is created for this task, which is depicted in code listing 3.

```
#!/bin/bash
if [ -z "$1" ]
then
  echo "please specify interface as first arg";
else
  # use airmon to stop interfering processes
  sudo airmon-ng check kill
  # then stop network manager
  # because airmon doesn't do a good job
  sudo service network-manager stop
  rfkill unblock wifi
  # Optionally kill other Wi-Fi clients the brute-for way:
  sudo pkill wpa_supplicant
  # Put the interface in monitor mode the old fashioned way
  sudo ifconfig $1 down
  sudo iwconfig $1 mode monitor
  sudo ifconfig $1 up
fi
```

Figure 3: Bash script that kills potentially interfering processes with `hostapd`.

After executing script 3 with the command `./prepare.sh wlan0`, the freshly compiled binary `hostapd` is launched with

```
./hostapd ../../wpa3.conf -dd -K
```

This command should create the WPA3 network using the device that was connected to the virtual machine. In this case, it was necessary to connect the network adapter Ralink Technology, Corp. RT5572 Wireless Adapter in bridged mode, such that the virtual machine guest operating system (Kali Linux) is able to directly access the physical device. It can be confirmed that the network is visible from remote supplicants by searching for the SSID **WPA3-Network** via an external Wi-Fi capable device such as a smartphone.

Instructions on the host: Connect to the Network via wpa_supplicant Now that the WPA3 access point is running in the virtual machine, a connection with wpa_supplicant is made. First wpa_supplicant needs to be compiled similar to the compilation instructions for hostapd.

```
wget https://w1.fi/releases/wpa_supplicant-2.8.tar.gz
tar xzvf wpa_supplicant-2.8.tar.gz
cd wpa_supplicant-2.8/wpa_supplicant
```

The line CONFIG_SAE=y needs to be appended to the file defconfig. Then wpa_supplicant is compiled with the commands

```
cp defconfig .config
make -j 2
cd ..
```

Then a WPA3-SAE capable configuration file **supplicant.conf** for the supplicant is created

```
network={
    ssid="WPA3-Network"
    psk="abcdefgh"
    key_mgmt=SAE
    #ieee80211w=2
}
```

All interfering processes must be terminated with the prepare.sh script depicted in figure 3. Then wpa_supplicant is started with the command

```
sudo ./wpa_supplicant -D nl80211 -i wlan0 -c supplicant.conf -K -dd
```

As can be verified from stdout, a successful SAE handshake is followed by association requests and responses and the 4-way handshake. The successful connection can be confirmed by checking the outputs from iwconfig on the host operating system:

```
iwconfig
wlan0     IEEE 802.11  ESSID:"WPA3-Network"
          Mode:Managed  Frequency:2.412 GHz  Access Point: 9E:0B:AE:5D:6D:A2
          Bit Rate=1 Mb/s   Tx-Power=15 dBm
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Power Management:on
          Link Quality=70/70  Signal level=-25 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:97   Missed beacon:0
```

4.3 Virtual Wi-Fi radios with mac80211_hwsim

To simulate virtual Wi-Fi devices, the Linux driver mac80211_hwsim may be used. The kernel module can be enabled with the command modprobe mac80211_hwsim and simulates an arbitrary number of 802.11 radio stations. Running user-space software such as hostapd or iwd with simulated interfaces from mac80211_hwsim does not differ significantly from hardware interfaces, because user-space programs cannot distinguish the simulated nl80211 and mac80211 kernel side implementation from a *real* one.

One major advantage of using modprobe mac80211_hwsim for fuzzing is the capability of programmatically creating 802.11 radios. From a practical standpoint, the possibility to launch and tear down Wi-Fi interfaces quickly saves a lot of time, which is indispensable when developing and debugging fuzzers. Even the configuration of hardware properties of the virtual 802.11 devices is possible. For example, its possible to enable support for protected management frames.

4.3.1 WPA3-SAE with mac80211_hwsim

First wpa_supplicant 2.8 and hostapd 2.8 must be installed as demonstrated in section 4.2. Then, the virtual 802.11 radio interfaces are created with the command modprobe mac80211_hwsim.

```
# kill all interfering daemons such as network-manager
sudo service network-manager stop
sudo pkill wpa_supplicant
# create 2 virtual 802.11 radios named wlan1, wlan2
sudo modprobe mac80211_hwsim radios=2
rfkill unblock wifi
# To monitor traffic in wireshark you can execute
sudo ifconfig hwsim0 up
# Set the channel of the radios to the same freq
sudo iwconfig wlan1 channel 1
sudo iwconfig wlan2 channel 1
```

The configuration files used for wpa_supplicant 2.8 and hostapd 2.8 are illustrated in figure 4a and 4b.

```
network={
        ssid="WPA3-Network"
        psk="abcdefgh"
        key_mgmt=SAE
        ieee80211w=2
}
```

(a) wpa_supplicant config file.

```
interface=wlan1
ssid=WPA3-Network
hw_mode=g
channel=1
wpa=2
wpa_passphrase=abcdefgh
wpa_key_mgmt=SAE
rsn_pairwise=CCMP
ieee80211w=2
sae_anti_clogging_threshold=0
```

(b) hostapd config file.

After the configuration files have been created, the WPA3 network is launched by executing the following commands in their respective directories:

```
# Open a new terminal, and in the hostapd directory execute:
sudo ./hostapd hostapd_wpa3.conf -dd -K
# Open another terminal, and in the directory wpa_supplicant execute:
sudo ./wpa_supplicant -D nl80211 -i wlan1 -c supp_wpa3.conf -dd -K
```

By executing the command `sudo ifconfig hwsim0 up`, traffic monitoring is enabled. This allows the observation of the complete WPA3-SAE handshake with a packet capturing tool such as Wireshark on the interface `hwsim0`. To filter the SAE authentication frames, the wireshark filter `wlan.fc.type_subtype==0x0b` can be used. Or even better, all frames except beacon frames can be displayed with the filter `wlan.fc.type_subtype!=0x08`.

4.3.2 Connecting `iwd` to `hostapd` using WPA3-SAE

In the following section, instructions how to connect the Intel Wireless daemon to `hostapd` are provided. `iwd v0.18`[8] is used as a supplicant and `hostapd v2.8`[9] is used as an authenticator/access point [Jou19; Int19]. Virtual radio interfaces must be created as was shown in section 4.3.1. This section assumes that `hostapd` uses the virtual interface `wlan3` and `iwd` used `wlan1`. The `hostapd` configuration that was used is depicted in figure 5. It uses the `sae_password` configuration variable instead of the `wpa_passphrase` variable as shown in section 4.3.1.

```
hw_mode=g
channel=1
ssid=ssidSAE
wpa=2
wpa_key_mgmt=SAE
wpa_pairwise=CCMP
sae_password=secret123|mac=ff:ff:ff:ff:ff:ff
ieee80211w=2
# SAE threshold for anti-clogging mechanism (dot11RSNASAEAntiCloggingThreshold)
# This parameter defines how many open SAE instances can be in progress at the
# same time before the anti-clogging mechanism is taken into use.
sae_anti_clogging_threshold=0
```

Figure 5: WPA3-SAE configuration file for `hostapd` when connected to `iwd`.

Then the following commands need to be executed to start `hostapd`, `iwd` and `iwctl`.

```
# TTY1: start hostapd with the above configuration
sudo ./hostapd/hostapd sae.conf -i wlan3 -K
# TTY2: launch iwd daemon with the correct physical interface
```

[8] iwd 0.18, accessed on 3th August 2019, `git://git.kernel.org/pub/scm/network/wireless/iwd.git`

[9] hostapd 2.8, accessed on 3th August 2019, `https://w1.fi/releases/hostapd-2.8.tar.gz`

```
# phy1 belongs to wlan1
sudo ./src/iwd --debug -p phy1
# TTY3: now start the iwd client
./client/iwctl
# use the following commands in iwctl CLI
# to connect to the "ssidSAE" network
>>> station list
>>> station wlan1 show
>>> station wlan1 disconnect
>>> station wlan1 scan
>>> station wlan1 get-networks
>>> station wlan1 connect ssidSAE
```

After having executed the above commands, `iwd` should have established a WPA3-SAE connection with the `hostapd` authenticator as can be verified in the respective terminal windows.

4.4 Remote Fuzzing

If the working machine is used as fuzzer and `Synology MR2200ac Router` as fuzzee, a real-world attack environment is obtained. The development of a fuzzer in such a environment has many drawbacks, since debugging access to proprietary routers is usually limited. Luckily, with the `Synology MR2200ac Router`, it is possible to login as `root` user to the BusyBox Linux operating system installed on the router.

4.4.1 Synology MR2200ac Router

The only router with WPA3-SAE support that was tested during this thesis, was the `Synology MR2200ac Router` with serial number `1910QLRHMA2HS`. This router supports WPA3-Personal and WPA3-Transition mode as well as WPA2 and WPA.

After establishing a wired connection via Ethernet, a SSH daemon with an user account with administrative login was created. A quick search revealed that the router uses `hostapd 2.7` and not the most recent version 2.8.

```
SynologyRouter> hostapd -v
hostapd v2.7-devel
[...]
```

It was not tested if the router automatically updates the firmware after connecting to the Internet. If this is not the case, then the `Synology MR2200ac Router` would be vulnerable against all the security vulnerabilities that have been published in Vanhoefs *dragonblood* paper. [VR19]

After configuring the router to use WPA3-Personal with the SSID `synologyWPA3`, the corresponding `hostapd` configuration file can be seen in figure 6.

```
SynologyRouter> cat /etc/hostapd/hostapd-wlan0-host.conf
interface=wlan0
driver=atheros
ssid=synologyWPA3
ignore_broadcast_ssid=0
ctrl_interface=/var/run/hostapd
max_num_sta=128
channel=1
wmm_enabled=1
preamble=1
bridge=lbr0
wpa_group_rekey=3600
hw_mode=g
ieee80211n=1
ht_capab=[SHORT-GI-20][DELAYED-BA][SHORT-GI-40][MAX-AMSDU-7935][HT40+]
auth_algs=1
wpa=2
wpa_key_mgmt=SAE
sae_password=letmein1234
wpa_pairwise=CCMP
ieee80211w=2
macaddr_acl=0
deny_mac_file=/etc/hostapd/hostapd-mac.list
uuid=c9a32f4e-d849-463f-ad62-e04c324274bf
wps_state=2
config_methods=label virtual_display virtual_push_button physical_push_button keypad
wps_rf_bands=ag
eap_server=1
upnp_iface=lbr0
friendly_name=Synology Router
```

Figure 6: Generated `hostapd` configuration for WPA3-Personal taken from the `Synology MR2200ac Router`.

The interfaces that are used by `hostapd 2.7` to establish the WPA3-Personal network are listed in figure 7. The interface listings have been obtained with the `iwconfig` command.

```
wlan0     IEEE 802.11ng  ESSID:"synologyWPA3"
          Mode:Master  Frequency:2.412 GHz  Access Point: 00:11:32:A5:36:E5
          Bit Rate:192 Mb/s   Tx-Power:16 dBm
          RTS thr:off   Fragment thr:off
          Encryption key:590F-4C04-1BC4-ED1E-247C-B967-171C-1107 [2]   Security mode:restricted
          Power Management:off
          Link Quality=94/94  Signal level=-97 dBm  Noise level=-95 dBm
          Rx invalid nwid:137  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0
```

Figure 7: The Wi-Fi interfaces powering the WPA3-Personal networks in the `Synology MR2200ac Router`

Elaborate logging is enabled in the router. This means that it can be easily checked which code paths of `hostapd 2.7` were reached. However, the standard loglevel includes no debugging information. Therefore, the upstart `/etc/init/hostap.conf` configuration file was updated to include the flags `-dd` and `-K` to increase the logging level and print cryptographic keys. Then the daemon was restarted with the command `restart hostap IFACE=wlan0` and increased debugging messages in the logfile could be obtained.

After this configuration modification of the `Synology MR2200ac Router`, `dragonfuzz.c` was launched against the router and all potential crashes were monitored via a separate SSH connection over Ethernet. The monitoring includes observations of the `hostapd` daemon, a kernel log and all restarts of related daemons.

4.5 Chosen Environment

In order to productively develop a fuzzing framework, a fuzzing environment with full control of the access point is indispensable. Controlling the access point allows to debug programs more efficiently and find bugs in reasonable time.

This can be done by launching a virtual software access point using `hostapd`. It is reasonable to use `hostapd` as an access point, since most professional routers either use `hostapd` directly or at least develop their own branch on top of it. In this thesis, `hostapd 2.8` was compiled on Ubuntu 18.04 and configured as WPA3-SAE access point as was shown in section 4.3. Protected management frames were not important, because they require support on the driver level. Furthermore, management frame encryption affects messages merely after a successful 4-way handshake.

4.5.1 Dragonfuzz

This thesis profited immensely from the research conducted in the paper *Dragonblood: A Security Analysis of WPA3's SAE Handshake* from Vanhoef et al. [VR19]. In their paper, Vanhoef et al. developed a tool named `dragondrain` [Van19] which implements a DoS attack by abusing the anti-clogging functionality implemented in SAE. A fuzzer named `dragonfuzz.c` on top of Vanhoef's GNU licensed source code was contributed in this thesis [Tsc19a]. *Dragondrain* and `dragonfuzz.c` are both implemented within the `aircrack-ng` framework, whose main purpose is to provide appropriate radiotap headers.

`dragonfuzz.c` is capable of a normal WPA3-SAE authentication exchange, including processing beacon frames from the targeted authenticator and sending a association request after the successful authentication. The first milestone in the development was to negotiate a correct WPA3-SAE authentication handshake. After this, it is possible to create stateful fuzzing tests that deliberately deviate from the correct handshake in the hope to trigger security vulnerabilities.

Limitations of `dragonfuzz.c` are:

1. The fuzzing program only implements WPA3-SAE with elliptic curves, because multiplicative groups are not activated by default in most WPA3-SAE implementations [VR19; Jou19].
2. The subsequent 4-way handshake is not fuzzed, because such work has already been exhaustively done by [VP18b] [VSP17] and others. The reason is that the 4-way handshake is not modified on a framing level when using SAE authentication, only the input PMK has higher entropy.

5 WPA3-SAE Model

In order to write fuzzing tests for the WPA3-SAE handshake, a model of the handshake needs to be derived from existing literature and standards.

The approach to derive the model follows a hybrid strategy. Tutorials and introductions in published papers such as [VSP17] and [VR19] are considered. Furthermore, existing standards such as the Dragonfly RFC [Har15] and even IEEE discussions in Internet forums[10] are parsed for relevant information. Unfortunately, the official IEEE Std 802.11 Wi-Fi standard [IEE16] is not open for public review and must be purchased for around 1.100 USD[11]. Luckily, the relevant parts of the standard could be obtained without purchasing the full product (For example from source code comments found in `hostapd` [Jou19]). Security research would greatly benefit from a more accessible standard.

As a further source of information, the WPA3 reference implementation in `wpa_supplicant` and `hostapd` and `iwd` are analyzed. The code analysis yields a state machine of the possible authentication frames exchanged during the WPA3-SAE handshake. It was decided to only consider the authentication frames of the Wi-Fi connection establishment process in WPA3-Personal. This means that the fuzzing framework will focus merely on the Auth-Commit and Auth-Confirm frame.

In a first step, a general classification of vulnerability types is introduced. The chapter concludes with a collection of fuzzing test cases that are considered to be interesting for further investigation.

As can be seen in the sequence diagram in figure 8, a Wi-Fi station initializes the WPA3-SAE handshake by sending an Auth-Commit frame to another station. If the frame is valid, the station replies with an Auth-Commit frame and an Auth-Confirm frame. If the exchange so far was correct, the station proceeds with an Auth-Confirm frame and the WPA3-SAE authentication handshake is completed. At this stage, both the station and access point share a common pairwise master key (PMK).

After the WPA3-SAE authentication, the station sends an association request with the cipher that was advertised in the preceding beacon frames and probe response frames (which are depicted in the beginning of the sequence diagram of figure 8).

After a successful association, the 4-way handshake follows. The PMK derived in the WPA3-SAE authentication exchange is used as input to the 4-Way handshake (instead of the PSK, which usually is a low entropy shared password). Because the PMK is a high entropy string and is updated every time a new WPA3-SAE authentication is performed, the 4-way handshake is immune against offline dictionary attacks and has

[10] IEEE 802.11 discussion about SAE, accessed on 21th August 2019, `https://mentor.ieee.org/802.11/documents?is_dcn=SAE`

[11] See prizing for 802.11 standard, accessed on 21th August 2019, `https://www.techstreet.com/ieee/standards/ieee-802-11-2016?product_id=1867583`

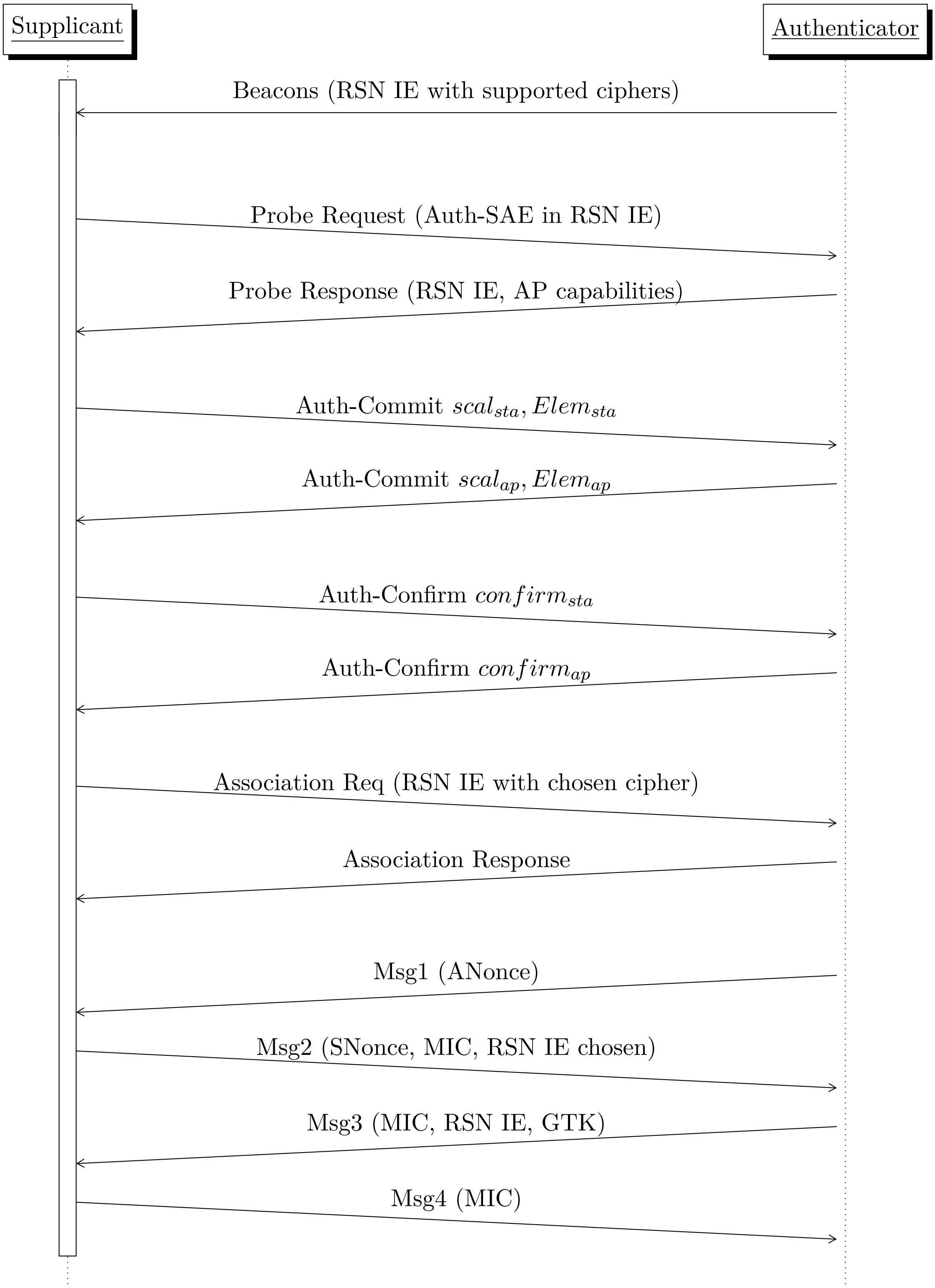
Supplicant
Authenticator
Beacons (RSN IE with supported ciphers)
Probe Request (Auth-SAE in RSN IE)
Probe Response (RSN IE, AP capabilities)
Auth-Commit $scal_{sta}, Elem_{sta}$
Auth-Commit $scal_{ap}, Elem_{ap}$
Auth-Confirm $confirm_{sta}$
Auth-Confirm $confirm_{ap}$
Association Req (RSN IE with chosen cipher)
Association Response
Msg1 (ANonce)
Msg2 (SNonce, MIC, RSN IE chosen)
Msg3 (MIC, RSN IE, GTK)
Msg4 (MIC)

Figure 8: The complete model of the WPA3-SAE handshake, including announcing beacons frames, probe request and probe response frames, association request frames and the finalizing 4-way handshake [VR19].

perfect forward secrecy.

WPA3-SAE is a protocol that may be initiated by both participants at the same time, as the name Simultaneous Authentication of *Equals* implies. However, in this thesis, only the common scenario where a 802.11 station connects to an 802.11 access point is considered. In this configuration, the station sends an initial Auth-Commit frame and receives the commit and confirm frame from the access point at the same time. This infrastructure resembles more a client-server architecture instead of a mesh architecture. [VR19]

5.1 Vulnerability Taxonomy

In this section, various important vulnerability classes that *could* affect the WPA3-SAE handshake are presented.

1. Programming errors that lead to memory corruption vulnerabilities such as stack and heap overflows, signed and unsigned integer overflows, null dereference errors, format string vulnerabilities, off-by-one errors and so on
2. Timing attacks or side channel attacks
3. Cryptographic implementation mistakes
4. Denial of service attacks
5. Other logical mistakes

A model based fuzzer will most likely not be able to trigger logical vulnerabilities such as timing attacks or side channel attacks. Usually, a fuzzing engine triggers memory corruption vulnerabilities caused by programming errors in low level C/C++ code. From a fuzzing strategy, all frames that have variable length fields or optional values require complex parsing logic that may lead to implemention mistakes causing memory corruption vulnerabilities.

5.2 Fuzzing Policy

A classic blackbox fuzzer does not care about specific protocols or protocol states. It rather modifies frame contents according to predefined heuristics and observes if the program crashes. An advantages is the straightforward implementation. Blackbox fuzzers are not protocol aware and development resources are thus relatively low.

However, such as blackbox approach brings a few disadvantages with it: The generated frames won't pass the first rudimentary checks in the parsing functions, resulting in low code coverage reached by the fuzzed inputs. The probability that randomly generated frames are in the correct format is tiny. Furthermore, no state awareness is possible.

Therefore, a better approach is to create a protocol aware fuzzer that only modifies one field at the time, while keeping all other frame fields static. If such an modification triggers an error, it is obvious which alteration caused it. This fuzzing policy allows to systematically fuzz all relevant program paths.

5.3 WPA3-SAE Framing

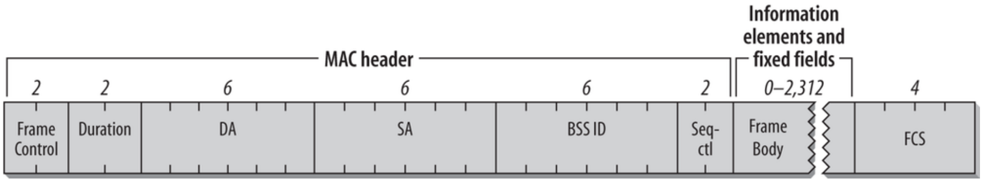

Figure 9: Format of a generic 802.11 management frame [Gas05].

802.11 authentication frames are a subtype of 802.11 management frames. The format of a generic management frame is illustrated in figure 9. There exist roughly a dozen different management frames used in the 802.11 standard to manage and control Wi-Fi networks. The actual data is sent in 802.11 data frames. In figure 10, an overview of different management frames is given without explicitly elaborating each frame type in detail.

```
/* 802.11-2016, Table 9-1 "Valid type and subtype combinations" */
enum mpdu_management_subtype {
        MPDU_MANAGEMENT_SUBTYPE_ASSOCIATION_REQUEST    = 0x0,
        MPDU_MANAGEMENT_SUBTYPE_ASSOCIATION_RESPONSE   = 0x1,
        MPDU_MANAGEMENT_SUBTYPE_REASSOCIATION_REQUEST  = 0x2,
        MPDU_MANAGEMENT_SUBTYPE_REASSOCIATION_RESPONSE = 0x3,
        MPDU_MANAGEMENT_SUBTYPE_PROBE_REQUEST          = 0x4,
        MPDU_MANAGEMENT_SUBTYPE_PROBE_RESPONSE         = 0x5,
        MPDU_MANAGEMENT_SUBTYPE_TIMING_ADVERTISEMENT   = 0x6,
        MPDU_MANAGEMENT_SUBTYPE_BEACON                 = 0x8,
        MPDU_MANAGEMENT_SUBTYPE_ATIM                   = 0x9,
        MPDU_MANAGEMENT_SUBTYPE_DISASSOCIATION         = 0xA,
        MPDU_MANAGEMENT_SUBTYPE_AUTHENTICATION         = 0xB,
        MPDU_MANAGEMENT_SUBTYPE_DEAUTHENTICATION       = 0xC,
        MPDU_MANAGEMENT_SUBTYPE_ACTION                 = 0xD,
        MPDU_MANAGEMENT_SUBTYPE_ACTION_NO_ACK          = 0xE,
};
```

Figure 10: An overview of management frame types and their identifiers as defined in `iwd v0.18`. [Int19]

The generic frame format of authentication frames is depicted in figure 11.

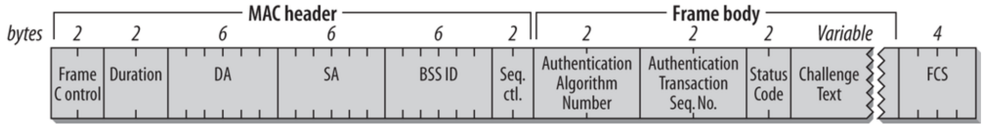

Figure 11: Format of a 802.11 authentication frame [Gas05].

The frame control field in the MAC header specifies the type of management frame. In the case of authentication frames, the allotment of the frame control field is illustrated in figure 12. The radiotap header is not included, since its display is merely an analytical help provided by the mac80211 sublayer and 802.11 drivers for packet inspection tools such as Wireshark. Furthermore, the MAC addresses in figure 12 are artificial and originate from a `hwsim` simulated network as described in section 4.3.

```
IEEE 802.11 Authentication, Flags: ........
    Type/Subtype: Authentication (0x000b)
    Frame Control Field: 0xb000
        .... ..00 = Version: 0
        .... 00.. = Type: Management frame (0)
        1011 .... = Subtype: 11
        Flags: 0x00
            .... ..00 = DS status: Not leaving DS or network is
             operating in AD-HOC mode (To DS: 0 From DS: 0) (0x0)
            .... .0.. = More Fragments: This is the last fragment
            .... 0... = Retry: Frame is not being retransmitted
            ...0 .... = PWR MGT: STA will stay up
            ..0. .... = More Data: No data buffered
            .0.. .... = Protected flag: Data is not protected
            0... .... = Order flag: Not strictly ordered
    .000 0001 0011 1010 = Duration: 314 microseconds
    Receiver address: 02:00:00:00:03:00 (02:00:00:00:03:00)
    Destination address: 02:00:00:00:03:00 (02:00:00:00:03:00)
    Transmitter address: 02:00:00:00:00:00 (02:00:00:00:00:00)
    Source address: 02:00:00:00:00:00 (02:00:00:00:00:00)
    BSS Id: 02:00:00:00:03:00 (02:00:00:00:03:00)
    .... .... .... 0000 = Fragment number: 0
    0000 0000 0001 .... = Sequence number: 1
```

Figure 12: The MAC header of a authentication frame taken from a Wireshark packet capture. The Wi-Fi network was created as described in section 4.3.

The authentication process usually involves a number of steps that depend on the specific authentication algorithm. The sequence number in authentication frames orders the steps of the authentication exchange. The status code and challenge text are used in different ways corresponding to the authentication algorithm used [Gas05].

It is crucial to understand that there are two sequence numbers of two octets size in an authentication management frame: One sequence number in the MAC header that increases with every 802.11 frame sent and one authentication sequence number in the frame body that specifies the current step in the authentication exchange.

The WPA3-SAE authentication algorithm uses a status code number 3. The authentication sequence number is 1 if its an Auth-Commit frame and 2 if it is a Auth-Confirm frame. The status code of the authentication frame depends on the sequence number. A status code of 0 indicates success. There exist a number of different status codes for WPA3-SAE authentication and an overview is given in figure 13 for `iwd 0.18` and `wpa_supplicant 2.8`.

It is particularly noteworthy that the status code `UNKNOWN_PASSWORD_IDENTIFIER` defined in `hostapd` is left undefined in `iwd`. This potentially indicates a difference in frame parsing and thus room for potential vulnerabilities. The status code that specifies a failed verified confirm message is named `CHALLENGE_FAILURE`.

```
/*
 * Program: iwd 0.18
 * Source: iwd/src/mpdu.h
 * 802.11-2016, Section 9.4.1.9 Status Code field
 */
enum mmpdu_status_code {
        MMPDU_STATUS_CODE_SUCCESS = 0,
        MMPDU_REASON_CODE_UNSPECIFIED = 1,
        MMPDU_STATUS_CODE_ANTI_CLOGGING_TOKEN_REQ = 76,
        MMPDU_STATUS_CODE_UNSUPP_FINITE_CYCLIC_GROUP = 77,
};
/*
 * Program: hostapd 2.8
 * Source: src/common/ieee802_11_defs.h
 * Status codes (IEEE Std 802.11-2016, 9.4.1.9, Table 9-46) */
define WLAN_STATUS_SUCCESS 0
define WLAN_STATUS_ANTI_CLOGGING_TOKEN_REQ 76
define WLAN_STATUS_FINITE_CYCLIC_GROUP_NOT_SUPPORTED 77
define WLAN_STATUS_UNKNOWN_PASSWORD_IDENTIFIER 123
```

Figure 13: SAE-Authentication status codes from `iwd` and `hostapd` [Int19; Jou19].

5.3.1 The Auth-Commit Frame

The structure of a Auth-Commit frame depends on various factors. As of June 2019, there is a ongoing discussion on frame formats in IEEE working groups, such as the group TGm[12].

For example, the document from Daniel Harkins uploaded on the 13th March 2019 with the document number 387[13] addresses various issues regarding a lack of clarity of

[12] TGm IEEE group discussion, accessed on 15th August 2019, `https://mentor.ieee.org/802.11/documents?is_dcn=sae`

[13] Document number 387, accessed on 15th August 2019, `https://mentor.ieee.org/802.11/dcn/19/`

the standard. The document with the title *Fixing some SAE issues* contains a similarly interesting discussion[14].

From the discussion of those documents, it is obvious that the programmers that implement the SAE protocol do not fully understand how Auth-Commit and Auth-Confirm frames are constructed and parsed. This discussion is very interesting to the matter of this thesis, because it allows the derivation of fuzzing tests based on expressed ambiguities. Especially when considering the fact that there are already routers that support WPA3-SAE released to the market months before this discussion (Example: `Synology Mesh Router MR2200ac`), it is questionable whether vendors implemented the standard in a consistent and correct manner.

The most important ambiguities occurring in the documents are quoted in the following paragraphs:

`CID 2590` states that *"The Password Identifier element is included in the unprotected authentication frame. It may violate the privacy of users (household). For example, it exposes a group of devices and number of devices that are sharing the same password. Particularly, when these devices belongs to the same household (apartment) in an apartment building, it violates the privacy of users/residents."* [Har19d]

and `CID 2690` informs that *"In SAE when Password Identifier is used, STA sends Password Identifier in the clear in auth frame. Since the Password Identifier is typically the identifier of a long-term password, the same Password Identifier would be sent each time a STA performs SAE authentication with a given network, and this could be used by bad actors to track a user's location with privacy implications."* [Har19d] with the proposal of the authors to *"during each SAE auth/assoc procedure, the AP can securely provision STA with a randomly-generated pseudonym for the SAE Password Identifier which the STA uses on the next SAE auth/assoc with that AP."* [Har19d]

SAE allows the usage of an **Password Identifier (PI)** that maps a confidential password to an unconfidential identifier. The idea is to provide the authentication algorithm with an clear text identity, such that the authenticator knows which entity attempts to authenticate. The key idea is that Wi-Fi stations only need to send the identifier and both the authenticator and supplicant immediately know which password should be used in the password derivation algorithm and subsequent handshake.

The concerns revolving around the PI are of privacy nature. An passive observer can learn what devices belong to which person, when he observes that the authentication uses the same password identifier among different devices. Furthermore, the geographical movements of an user may be tracked when the user authenticates with PIs [Har19d].

Daniel Harkins rejects the issue with the remark that the MAC address of the stations carries the same information regarding assignability and that no new attack vectors

`11-19-0387-02-000m-addressing-some-sae-comments.docx`

[14] Document number 733, accessed on 15th August 2019, `https://mentor.ieee.org/802.11/dcn/19/11-19-0733-00-000m-fixing-some-sae-issues.docx`

would be opened [Har19d]. Furthermore, he strictly rejects the removal of the PI functionality, because *"it is not mandatory to use"* and it is an *"useful feature"* [Har19d]. However, Harkin endorses the idea of obfuscating the password identifier in `CID 2690`, but notes that there might be issues managing the pseudonyms when the number of users grows that share the same PI [Har19d].

There is another request in document `CID 2546` that proposes to transform the scalar, elements and anti-clogging token into information elements, such that parsing follows the type-length-value logic and does not depend on fixed sizes. Harkins rejects the issue with the remark that *"parsing is based on the length of components and not their contents"* [Har19d]. However, the last part of his statement is not entirely true in practical implementations, because the parsing of the password identifier is based on the contents of the three first bytes of the password identifier, as the code listing from `hostapd` in figure 14 proves.

```
static int sae_is_password_id_elem(const u8 *pos, const u8 *end)
{
        return end - pos >= 3 &&
                pos[0] == WLAN_EID_EXTENSION &&
                pos[1] >= 1 &&
                end - pos - 2 >= pos[1] &&
                pos[2] == WLAN_EID_EXT_PASSWORD_IDENTIFIER;
}
```

Figure 14: Source code of the parsing of password identifiers in `hostapd 2.8`. [Jou19]

The framing format of a Auth-Commit frame is depicted in table 2.

Auth Algorithm No (2 Bytes)	Auth Transaction Seq No (2 Bytes)	Status Code (2 Bytes)
Group Id (2 Bytes)	Scalar (32 Bytes)	
Element X-coordinate (32 bytes)	Element Y-coordinate (32 bytes)	

Table 2: Auth-Commit frame without anti-clogging token and password identifier.

By inspecting the code of `hostapd 2.8` and especially the function `sae_parse_commit_token()`, an optional password identifier can be located after the scalar and element or after the scalar and element and anti-clogging token, if a optional anti-clogging token is present. The anti-clogging token in `hostapd 2.8` is usually 32 octets long, which is the length of a HMAC_SHA256 digest.

Furthermore, when multiplicative groups mod p are used instead of ECC, the length of the scalar and element is $2 \cdot |p|$ instead of $3 \cdot |p|$, because in ECC, the element consists of an point with two coordinates. Therefore, $|p|$ is different among ECC and multiplicative gropus, where the former is usually up to 10 times smaller [PP09].

In table 3 below, a commit frame with a optional anti-clogging token and an optional password identifier is illustrated. The password identifier may be located before or after the anti-clogging token.

Auth Algorithm No (2 Bytes)	Auth Trans Seq (2 Bytes)	Status Code (2 Bytes)
Group Id (2 Bytes)	Scalar (32 Bytes)	
Element X-coordinate (32 bytes)	Element Y-coordinate (32 bytes)	
Optional Password Identifier (1 to 254 Bytes)		
Anti-Clogging-Token (32 to 256 Bytes)		
Optional Password Identifier (1 to 254 Bytes)		

Table 3: Auth-Commit frame with optional anti-clogging token and optional password identifier present. The password identifier can be located before or after the anti-clogging token.

The password identifier must have a preamble with size of three octets, where the first octet is set to `WLAN_EID_EXTENSION` with value 255, the second octet is a length value that must be at least 1 and maximally 254, and the third octet is set to `WLAN_EID_EXT_PASSWORD_IDENTIFIER` with value 33.

Request Anti-Clogging Token Frame A special commit message is constructed when the access point has anti-clogging tokens enabled as defense mechanism against DoS attacks. If a station sends an Auth-Commit frame to an access point and this access point has anti-clogging tokens enabled, the access point will answer with an anti-clogging token in a Auth-Commit frame with a format as illustrated in figure 4. The status code must be set to `WLAN_STATUS_ANTI_CLOGGING_TOKEN_REQ`.

Auth Algorithm No (2 Bytes)	Auth Trans Seq (2 Bytes)	Status Code (2 Bytes)
Group Id (2 Bytes)	Anti-Clogging-Token (32 to 256 Bytes)	

Table 4: Frame format when an access point requests an anti-clogging token from the supplicant.

Invalid Group Commit Frame An authenticator shall reject commit frames when the supplicant initiates with an unsupported group id. This is achieved by replying to an Auth-Commit frame with another Auth-Commit frame with status `WLAN_STATUS_FINITE_CYCLIC_GROUP_NOT_SUPPORTED`. This request-response cycle continues as long as the authenticator does not run out of alternative groups.

5.3.2 The Auth-Confirm Frame

The Auth-Confirm frame is much simpler than the Auth-Commit frame. It consists of two fields: A send confirm number of two octets size and a confirm token that is 32 octets long. In the confirm exchange, both authentication participants confirm that they derived the same secret and are in possession of the same password [Har15].

Auth Algorithm No (2 Bytes)	Auth Trans Seq (2 Bytes)	Status Code (2 Bytes)
Send Confirm (2 Bytes)	Confirm Token (32 Bytes)	

Table 5: The message structure of an Auth-Confirm frame.

The confirm token consists of a HMAC-SHA256 hash using the key confirmation key (kck) as hashing key and the send confirm number, the own scalar and own element followed by the peer scalar and peer element as concatenated input message. In order to verify the Auth-Confirm token of an authenticator, the supplicant uses the send confirm number of the authenticator, the authenticator's scalar and element and its own scalar and element as inputs for the hash construction. The resulting hash is also called a *verifier*. The computation of the confirm token is illustrated in figure 15 as comment from `hostapd` [Jou19].

```
/* Confirm

* CN(key, X, Y, Z, ...) =
*    HMAC-SHA256(key, D2OS(X) || D2OS(Y) || D2OS(Z) | ...)

* confirm = CN(KCK, send-confirm, commit-scalar, COMMIT-ELEMENT,
*              peer-commit-scalar, PEER-COMMIT-ELEMENT)
* verifier = CN(KCK, peer-send-confirm, peer-commit-scalar,
*               PEER-COMMIT-ELEMENT, commit-scalar, COMMIT-ELEMENT)
*/
```

Figure 15: Computation of the Auth-Confirm token as conducted in `hostapd`. [Jou19]

What exactly is the purpose of two octet long send confirm number that is part of the confirm frame?

The send confirm number in the confirm exchange is a counter that tracks the number of confirm frames being sent. Every time a SAE authentication participant transmits a confirm frame, the send confirm number is incremented. When checking a confirm frame, the value of the send confirm number is compared against the received confirm number. The received confirm frame is verified only if the send confirm number is set to $2^{16} - 1$ and a own send confirm frame has been sent. This logic can be verified in source code listing 16. Each participant has a variable for the own send confirm number and the peer send confirm number.

```
sc = l_get_le16(frame);

/*
* ... the value of send-confirm shall be checked. If the value is not
* greater than Rc or is equal to 2^16 - 1, the received frame shall be
* silently discarded.
*/
if (sc <= sm->rc || sc == 0xffff)
        return false;

/*
* If the verification fails, the received frame shall be silently
* discarded.
*/
if (!sae_verify_confirm(sm, frame))
        return false;

/*
* If the verification succeeds, the Rc variable shall be set to the
* send-confirm portion of the frame, the Sync shall be incremented and
* a new SAE Confirm message shall be constructed (with Sc set to
* 2^16 - 1) and sent to the peer.
*/
sm->sync++;
sm->sc = 0xffff;

sae_send_confirm(sm);
```

Figure 16: The send confirm number is set to $2^{16} - 1$ when the received confirm frame was successfully verified, source code obtained from `iwd`. [Int19]

5.4 Finite State Machine

Now that the framing of the WPA3-SAE authentication has been introduced, a finite state machine of all possible state transitions as described in the standard **IEEE Std 802.11-2016** can be derived [IEE16]. Those state transitions give valuable insight and attacking points for potential fuzzing strategies.

The states in WPA3-SAE model are the following [IEE16]:

1. **Nothing**: The initial state of a new protocol execution.
2. **Committed**: In this state, the finite state machine has sent an Auth-Commit frame and is waiting for an Auth-Commit frame and Auth-Confirm frame from the peer.
3. **Confirmed**: The finite state machine sent an Auth-Commit and Auth-Confirm frame and is waiting for the Auth-Confirm frame of the peer.
4. **Accepted**: The protocol execution has been successfully terminated after receiving an Auth-Commit and Auth-Confirm frame.

Based on the standard, a state transition table for the SAE finite state machine may be created. According to the standard, implementations need to take care of those 22 possible states referenced in table 6 [IEE16].

The left most column of table 6 specifies the state transition. The middle column defines the events and conditions that trigger the state transition. The right column defines the action and consequences that a standard compliant WPA3-SAE implementation should implement.

SAE implementations have a retransmission timer for WPA3-SAE frames and a PMK expiration timer. When the PMK expiration timer times out, a new handshake execution is initiated. A timeout for the retransmission timer causes a retransmission for the frame in question. Furthermore, implementations manage three counter variables. A *sync* variable that keeps track of all resynchronizations that have occurred. A *send confirm* and *receive confirm* counter that keep track of the number of confirm frames sent to the station itself and the peer. [IEE16]

Furthermore, the executing SAE process maintains a *database of protocol instances* [IEE16], secrets for anti-clogging tokens and counters for committed and confirmed states for the individual protocol instances. [IEE16]

State Transition	**Events / Conditions**	**Behavior**
Nothing → Nothing	Recv commit frame with bad group	Reply with commit frame with status 77, delete protocol state
Nothing → Nothing	Recv commit frame with status 76 or 77	Delete protocol state
Nothing → Committed	SAE authentication initiated by user or program	Set sync to 0, set send confirm to 0, set recv confirm to 0, send commit frame with success status, reset retransmission timer
Nothing → Confirmed	Recv commit frame with status 0 and valid group	Set sync to 0, set send confirm to 0, set recv confirm to 0, increment send confirm, send commit frame with success status, send confirm frame with success status, reset retransmission timer
Committed → Committed	Recv commit frame with invalid group, the number of state synchronizations have not exceeded the threshold	Reply with commit frame with status 77, increment the synchronization counter, reset retransmission timer
Committed → Committed	Recv commit frame with rejected group, but there are more groups to chose from	Set the synchronization counter to 0, reply with commit frame with success status, reset retransmission timer

State Transition	Events / Conditions	Behavior
Committed → Committed	Recv confirm frame with rejected group, the number of state synchronizations have not exceeded the threshold	Increment the synchronization counter, reply with commit with success status, reset retransmission timer
Committed → Committed	Retransmission timer fired and the number of state synchronizations have not exceeded the threshold	Increment the synchronization counter, reply with commit with success status, reset retransmission timer
Committed → Committed	Recv commit frame with a supported group that differs from the one offered, peer has a numerically smaller MAC address	Reply with commit frame with success status, reset retransmission timer
Committed → Committed	Recv commit with invalid anti-clogging token	Set the synchronization counter to 0, reply with commit with success status, reset retransmission timer
Committed → Confirmed	Recv commit frame with valid group and a group that was offered	Increment the send confirm counter, reply with confirm frame with status success, reset retransmission timer
Committed → Confirmed	Recv commit with valid group that differs from the group proposed, peer has numerically higher MAC address	Set the synchronization counter to 0, increment the send confirm counter, reply with commit frame with status success, then send confirm frame with success status, reset the retransmission timer
Confirmed → Nothing	Synchronization counter is too large	Delete protocol state
Confirmed → Confirmed	Recv commit frame and synchronization counter is not too large	Increment the send confirm number, increment synchronization counter, send commit frame with success status, send confirm frame, reset the retransmission timer
Confirmed → Confirmed	Recv commit frame with invalid anti-clogging token	Reset retransmission timer
Confirmed → Confirmed	Retransmission timer fired, synchronization counter is not too large	Increment the send confirm number, increment synchronization counter, reply with confirm frame, reset the retransmission timer
Confirmed → Accepted	Recv confirm frame and the confirm token is valid	Reset the PMK expiration timer

State Transition	Events / Conditions	Behavior
Accepted → Accepted	Recv confirm frame, the recv confirm number is not too large, synchronization counter is not too large	Increment synchronization counter, reply with confirm frame
Accepted → Accepted	Recv confirm frame, authentication is valid, synchronization counter is not too large	Increment synchronization counter, reply with confirm frame
Accepted → Nothing	PMK expiration timer timed out	Delete protocol state
Accepted → Nothing	Synchronization counter is too large	Delete protocol state

Table 6: The state transition table of WPA3-SAE. States have been inferred from the IEEE Std 802.11 2016 standard [IEE16].

5.5 Fuzzing Test Cases

In the previous sections, the framing of the WPA3-SAE authentication exchange has been treated in-depth, such that fuzzing strategies can be proposed now. Broadly speaking, fuzzing tests will either target the processing and parsing functionality of the Auth-Commit or Auth-Confirm frame. Furthermore, all fuzzing test cases are executed within a supplicant/authenticator infrastructure: The fuzzer is the supplicant, the fuzzee is the authenticator.

Even though the DoS vulnerability in `iwd 0.18` was discovered during a manual code audit (Compare section 6.2), it could have been easily found by a fuzzer that creates anti-clogging token commit frames with variable sized anti-clogging tokens. Therefore, one common fuzzing strategy is to alter the size of one field and keep all other fields of static size.

An complete overview of all implemented fuzzing test cases and their test case descriptions are presented in table 7. Most fuzzing test cases target the initial Auth-Commit frame and test a single path of failure in access point implementations. This helps enormously with the tracing of potentially triggered memory corruption errors. Only three of all fuzzing cases are stateful. However, the number of possible fuzzing mutations is large and amounts for all interesting corner cases.

The implementation of the fuzzing test cases illustrated in table 7 can be obtained from [Tsc19a] and [Tsc19c].

Fuzzing Test Name	**Fuzzing Test Description**
FUZZ AUTHENTICATION FRAME	Create a authentication frame were only the authentication method is set to 3 (SAE). The status and sequence authentication fields are fuzzable. Add a random fuzzable payload of data. The recipient is expected to ignore or reject the frame.
INVALID ELEMENTS AND SCALARS	Create a Auth-Commit frame with a scalar and elements set to all zero. Use a element that is the point at infinity in the ECC group. Use scalars and elements that are outside the allowed ranges.
REFLECT SCALAR AND ELEMENT	Upon receiving a Auth-Commit frame, reflect the scalar and element that were received from the peer. Peer is expected to abort the handshake.
RANDOM GROUP	Pick a random, but valid group in a otherwise valid Auth-Commit frame. The commit frame is expected to be rejected if the group is not supported.
RANDOM SCALAR	Fuzz a random but valid scalar in a otherwise correct Auth-Commit frame. The handshake is expected to fail after checking the confirm token.
RANDOM ELEMENT	Fuzz a random but valid element in a otherwise correct Auth-Commit frame. The handshake is expected to fail after checking the confirm token.
INVALID LENGTH	Inject a random sized blob of bytes after a otherwise valid Auth-Commit frame. The authentication is expected to either proceed without errors (because implementations ignore excess bytes) or fail.
RANDOM TOKEN	Add a random anti-clogging token without receiving a prior anti-clogging token request. The token check is expected to be ignored.
INVALID ELEMENT	Add random bytes as the element in the Auth-Commit frame. The construction of an group element is expected to fail on the receiving side.
RANDOM PASSWORD IE	Inject a random fixed sized password identifier at the end of a otherwise valid Auth-Commit frame. The password identifier is expected to be rejected.
VARIABLE TOKEN	Inject a commit frame with variable sized anti-clogging token from length 0 to 500.
VARIABLE PASS IDENTIFIER	Inject a Auth-Commit frame with random and variable sized password identifier with length between 0 to 255. Inject multiple password identifiers at different locations.
ALL STATUS CODES	Inject a valid Auth-Commit frame and enumerate all authentication status codes while the authentication algorithm field remains SAE (3) and sequence number 1 (Auth-Commit). Add a random sized payload with random data after a valid group, scalar and elements. The Auth-Commit frame is expected to be rejected, because the status code is nonzero (no success)

Fuzzing Test Name	**Fuzzing Test Description**
SEND INITIAL CONFIRM FRAME	Send a valid Auth-Confirm frame instead of beginning the authentication with a Auth-Commit frame. The recipient is expected to reject the Auth-Confirm frame.
TRIGGER ANTI-CLOGGING, DIFFERENT SCALAR/ELEMENTS	**Stateful fuzzing.** Trigger anti-clogging threshold at the access point. Then reply with a Auth-Commit frame with correct anti-clogging token set, but then continue fuzzing the group, elements, scalar and password identifier. The recipient is expected to reject the Auth-Commit frame.
VALID AUTH-COMMIT, FUZZED AUTH-CONFIRM PAYLOAD	**Stateful fuzzing.** Send a valid Auth-Commit frame and wait for the Auth-Commit response from the peer. Then fuzz all fields in the corresponding Auth-Confirm frame. The recipient is expected to reject the Auth-Confirm frame.
VALID AUTH-COMMIT, FUZZED SAE AUTHENTICATION FRAME	**Stateful fuzzing.** Send a valid Auth-Commit frame and wait for the Auth-Commit response from the peer. Then fuzz the status and sequence fields in the authentication frame body. Set the authentication algorithm to 3 (SAE). Add a fuzzable random blob of data at the end.

Table 7: Fuzzing test cases implemented in the *Dragonfuzz Framework*. Not all fuzzing test cases are stateful. The *Dragonfuzz Framework* implementation can be obtained from [Tsc19a].

6 Results

6.1 Dragonfuzz

The hybrid strategy of this thesis yielded three different fuzzing approaches that together form the **Dragonfuzz Framework** [Nik19]. Summarized, a C program was created that allows to fuzz selected states in the SAE handshake (The fuzzing test cases can be inspected in table 7). Additionally, a Python program was developed that fuzzes the Auth-Commit frame using the `boofuzz` fuzzing framework. Furthermore, the greybox fuzzing engine `libFuzzer` was used to target selected functionality in the open source projects `hostapd` and `iwd` [Jou19; Int19]. The detailed instructions how to replicate the results from the Dragonfuzz Framework can be obtained from their respective Git repositories [Nik19].

The first result is the development of a blackbox fuzzing program named `dragonfuzz.c`[15] that is capable of remotely fuzzing access points with WPA3-Personal support [Tsc19a]. `dragonfuzz.c` runs on top of the `aircrack-ng` framework and makes use of boilerplate code that Vanhoef et al. contributed in the *Dragonblood* paper [VR19]. Cryptography support is provided by the open source library OpenSSL. `dragonfuzz.c` is capable of executing a complete WPA3-SAE authentication exchange with full ECC cryptography support, similar to a proper supplicant such as `wpa_supplicant` or `iwd`. Therefore, `dragonfuzz.c` is a fuzzing test case generator that targets selected states and configurations in the WPA3-SAE handshake. The software is written in C and consists of around 2500 lines of source code with comments included.

Furthermore, a Python fuzzing script named `dragonfuzz.py`[16] was developed on top of the `boofuzz`[17] fuzzing framework [Tsc19c]. `Boofuzz` is the successor of the famous `sulley` network fuzzing engine [Boo19] and can be considered a blackbox network-fuzzing framework. `dragonfuzz.py` complements the functionality provided by `dragonfuzz.c`: All fuzzing test cases that don't need complicated stateful behavior such as password derivation and cryptographic verification are handled by `dragonfuzz.py`. Using the modern `boofuzz` fuzzing framework enables quick blackbox fuzzing development without spending time reinventing a basic fuzzy string mutation engine. While `dragonfuzz.c` can be considered a test case generator for single, stateful fuzzing tests that all need to be implemented individually in a time consuming process, `dragonfuzz.py` follows a more traditional fuzzing strategy that aims to uncover low-hanging programming mistakes.

Put differently, `dragonfuzz.c` can fuzz individual states in the SAE handshake because it implements all necessary functionality to simulate a complete handshake. `dragonfuzz.py` on the other hand cannot derive a password element and only fuzzes the initial Auth-Commit frame. `dragonfuzz.py` never reaches all code paths in the

[15] `dragonfuzz.c`, accessed on 5th August 2019, `https://gitlab.com/NikolaiT/dragonfuzz/`
[16] `dragonfuzz.py`, accessed on 5th August 2019, `https://github.com/NikolaiT/dragonfuzz/`
[17] `boofuzz`, accessed on 20th August 2019, `https://boofuzz.readthedocs.io/en/latest/`

Auth-Confirm step, because it lacks cryptography support that is necessary to derive the password element. However, from a fuzzing coverage standpoint, the Auth-Confirm frame has limited relevance, because there are only two static sized fields in the Auth-Confirm frame that include potentially vulnerable parsing logic.

The third part of the *Dragonfuzz Framework* contains a greybox fuzzing campaign targeting two functions in the open source software `hostapd` and `iwd` which are responsible for parsing 802.11 authentication frames such as the Auth-Commit and Auth-Confirm frame [Nik19].

6.1.1 Tested Hardware and Software

Unfortunately, at the time of writing in August 2019, there was a sparse amount of hardware that actually supports WPA3-SAE [All19b]. Even though the WPA3-Personal certification was released in early 2018, the Wi-Fi industry manufacturing hardware has not yet adopted the new standard on a broad scale [AVM19]. Therefore, *Dragonfuzz* could only be tested against one WPA3 capable device and against one access point implementation with WPA3-SAE support. Vanhoef et al. tested another WPA3 capable device, but could not release the product name for confidentiality reasons [VR19].

Target of Evaluation	**Fuzzing Test Result**
`Synology MR2200ac Router`	All vulnerabilities from the *Dragonblood* paper apply [VR19], because the router uses `hostapd 2.7` if not updated manually. No additional vulnerabilities have been found with a remote fuzzing test with the *Dragonfuzz Framework*.
`hostapd 2.8 / wpa_supplicant 2.8`	No new vulnerabilities found when remotely tested with the blackbox fuzzing framework `dragonfuzz.py` and `dragonfuzz.c`.
`iwd 0.18` located in `iwd/src/sae.c` [Int19]	A DoS vulnerability was found in the handling of anti-clogging tokens during a manual code review. A detailed description can be found in section 6.2.
`sae_rx_authenticate()` located in `iwd/src/sae.c` in `iwd 0.18` [Int19]	Extensive fuzzing with `libFuzzer` and AddressSanitizer, MemorySanitizer and UndefinedBehaviorSanitizer. The fuzzing tests revealed a unsigned integer overflow. Detailed explanation of the approach in section 6.1.2.
`ieee802_11_mgmt()` located in `hostap/src/ap/ieee802_11.c` in `hostapd v2.9-devel` [Jou19]	Extensive fuzzing with `libFuzzer` and AddressSanitizer, MemorySanitizer and UndefinedBehaviorSanitizer and a large WPA3-SAE frame corpus. Detailed explanation of the approach in section 6.1.3. No critical vulnerabilities have been found.

Table 8: Overview of the fuzzed WPA3-Personal capable software and hardware. [Nik19]

As table 8 illustrates, the only WPA3-SAE hardware device tested was the `Synology`

```bash
#!/bin/bash

export CC='clang-8'
export CXX='clang++-8'

autoreconf -i

./configure CFLAGS='-O1 -fno-omit-frame-pointer
 -g -ggdb3 -fsanitize=address,fuzzer-no-link
-fsanitize=integer' --prefix=/usr CC=clang-8
--enable-tools --enable-debug --enable-asan

make all -j4

make check TESTS='unit/test-sae'
```

Figure 17: The bash instructions to compile `iwd 0.18` with `libFuzzer` support.

`MR2200ac Router` released in late 2018 by the company Synology. If the router is not automatically updated, it suffers from all security vulnerabilities published in [VR19]. No new security vulnerabilities could be found with the fuzzing framework *Dragonfuzz*.

Furthermore, the main testing target during this thesis was `hostapd 2.8` [Jou19]. `hostapd 2.8` fixed all security vulnerabilities demonstrated in the paper *dragonblood* [VR19]. Fuzzing with `dragonfuzz.c` and `dragonfuzz.py` did not yield any new security vulnerabilities. It is conjectured that most low hanging programming mistakes that can be triggered by a blackbox fuzzer have already been found, because `hostapd` has been extensively fuzzed with `libFuzzer` as part of the distributed cloud fuzzing project OSS-Fuzz from Google[18].

The `hostap` source code ships with a dedicated fuzzing test suite targeting the function `ieee802_11_mgmt()` in `src/ap/ieee802_11.c` which is responsible for parsing all kinds of 802.11 management frames. The fuzzing corpus includes multiple SAE frame examples for ECC and FFC.

6.1.2 Coverage-guided Greybox Fuzzing of iwd

The 802.11 authenticator `iwd 0.18` [Int19] was fuzzed with a coverage-guided greybox-fuzzing approach with `libFuzzer` which is part of the LLVM compiler toolchain. As depicted in table 8, the function `sae_rx_authenticate()` located in `iwd/src/sae.c` was extensively fuzzed. Figure 19 illustrates the `libFuzzer` entry point for the test driver implementation targeting `sae_rx_authenticate()`. The compile instructions for `iwd 0.18` with `libFuzzer` can be reviewed in figure 17, the instructions to build and link the fuzzing driver can be looked up in figure 18.

[18] `hostap` fuzzing with `libFuzzer`, accessed on 5th August 2019, `https://github.com/google/oss-fuzz/tree/master/projects/hostap`

```
fuzz_sae_LDFLAGS = -fsanitize=fuzzer,address,leak
 -fsanitize=integer -fsanitize=undefined
```

Figure 18: Flags for the fuzzing driver `fuzz_sae.c` that were added to iwd's `Makefile.am`.

As the code listing in figure 19 reveals, a valid 802.11 authentication header was prepended to the fuzzed input data in order to direct the fuzzer in the correct target function. No serious programming mistakes have been found using fuzzing efforts with `libFuzzer` with AddressSanitizer, MemorySanitizer and UndefinedBehaviorSanitizer and clang version `clang version 8.0.1`. The highest code coverage reached as indicated by `libFuzzer` was 210. However, `libFuzzer` detected an unsigned integer overflow in line 1043 in the source code file `src/sae.c`:

```
ret = sae_verify_packet(sm, L_LE16_TO_CPU(auth->transaction_sequence),
                                        L_LE16_TO_CPU(auth->status),
                                        auth->ies, len - 6);
```

The unsigned integer overflow occurs when the frame length is five or smaller. It is possible to manufacture a SAE frame which results in a unsigned integer overflow in the expression $len - 6$. However, a manual audit revealed that this overflow has no further consequences, since the only critical code that uses the overflowed len variable was fixed after reporting the DoS vulnerability explained in detail in section 6.2.

Even though `iwd` seems to have been thoroughly tested with unit tests and other test cases, no prior coverage guided fuzzing approaches have been observed. It would be a good decision to fuzz all critical code blocks that consume remotely tainted frames, similar as it is currently done by the `hostap` developers. A fuzzing testing strategy in `iwd` would be especially advisable in other Wi-Fi protocol implementations such as EAPOL or WPS, because the parsing complexity there is much higher.

6.1.3 Coverage-guided Greybox Fuzzing of hostapd

The 802.11 access point `hostapd v2.9-devel` [Jou19] has been submitted to an coverage-guided greybox fuzzing campaign with `libFuzzer` [Nik19].

As depicted in table 8, the function `ieee802_11_mgmt()` located in `src/ap/ieee802_11.c` has been subjected to a greybox fuzzing test. The fuzzing test driver implementation can be obtained from the corresponding Github repository[19]. This Github repository also includes the Makefile and make instructions used to compile and link `hostapd v2.9-devel` with `libFuzzer` and the required memory sanitizers.

The fuzzing corpus was compiled with SAE authentication frames recorded via Wireshark during a simulated WPA3 network, which was created according to description

[19] fuzz_sae_hostap, accessed on 11th August 2019, `https://github.com/NikolaiT/fuzz_sae_hostap`

```
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  struct test_data *td = l_new(struct test_data, 1);
  struct auth_proto *ap = test_initialize_fuzz(td);

  uint8_t *auth_mgmt_header = "\xB0\x00\x3A\x01\x02\x00\x00\x00\x00\x00\x02\x00"
                              "\x00\x00\x02\x00\x02\x00\x00\x00\x02\x00\x10\x04";

  size_t frameSize = 24;
  size_t bufSize = Size + frameSize;
  uint8_t fuzzData[bufSize];
  memset(fuzzData, 0, bufSize);
  memcpy(fuzzData, auth_mgmt_header, frameSize);
  memcpy(fuzzData + frameSize, Data, Size);
  auth_proto_rx_authenticate(ap, (uint8_t *)fuzzData, bufSize);
  test_destruct(td);
  auth_proto_free(ap);
  return 0;
}
```

Figure 19: `fuzz_sae`: The `libFuzzer` driver that targets `sae_rx_authenticate()` in `iwd`. [Int19]

in section 4.3. The maximum coverage reached during the fuzzing campaign was 346. No programming mistakes were uncovered in this fuzzing test.

The fuzzer was started using the command

```
./sae sae_corpus_2 -detect_leaks=0 -max_len=1050 -print_final_stats=1
```

and has been manually confirmed to reach the SAE processing and parsing functionality located in `src/common/sae.c`.

Fuzzing the function `ieee802_11_mgmt()` has several disadvantages. Since `ieee802_11_mgmt()` is placed relatively high in the call stack of `hostapd`, the function is responsible for handling all possible management frames in Wi-Fi networks. Therefore, the fuzzer might reach code paths unrelated to the handling of SAE frames.

Furthermore, `hostapd` handles SAE authentication messages in a queue with limited capacity. Each call to `auth_sae_process_commit()` is delayed linearly with the number of pending SAE authentication messages. Before fuzzing, this delaying mechanism had to be disabled manually.

An advantage of targeting a function high in abstraction is the increased likelihood to trigger logical flaws in the handling of SAE frames. Put differently, if a lower tier function such as `sae_parse_commit()` located in `src/common/sae.c` would have been chosen as a fuzzing entry point, much less code could be covered by the fuzzer.

Furthermore, the detection of memory leaks had to be disabled because it appears that `hostapd` does not properly free allocated memory when processing commit messages. Those tiny memory leaks accumulate quickly during a fuzzing campaign with many million executions and result in an abort panic after around five minutes.

6.2 Denial of Service Vulnerability in iwd

While the author of this thesis (Nikolai Tschacher, `nikolai@tschacher.ch`) manually audited the source code of the wireless daemon `iwd v0.18`[20] (iNet wireless daemon), a Denial Of Service vulnerability was located by him in the handling of anti-clogging tokens in `iwd/src/sae.c`. `iwd` is a potential modern replacement for `wpa_supplicant` with very few dependencies and low footprint [Int19]. `iwd` only makes use of the small ELL (Embedded Linux Library)[21] and does not depend on external cryptographic libraries such as Openssl or Wolfssl, because the Linux kernel crypto subsystem is used for all cryptographic operations. As a consequence, `iwd` runs only on Linux distributions and requires a recent kernel to work. `iwd` is a good fit for small embedded devices that need a 802.11 supplicant to communicate over radio. `iwd` is often used in repeaters, Internet-of-Things devices and all kinds of electronic household hardware with Wi-Fi support and industrial wireless devices.

The vulnerability is related to the handling of anti-clogging tokens in the WPA3-SAE authentication handshake. Ironically, the anti-clogging defense of WPA3-SAE tries to mitigate DoS attacks that arise when an attacker floods the victim with many forged commit frames which invoke a cascade of costly commit frame processing operations: Password element derivation, quadratic residue blinding and the mitigations against side channel attacks and timing attacks themselves [VR19].

The anti-clogging mechanism consists of a simple cookie exchange procedure. When the anti-clogging defense is activated in the access point, upon reception of an Auth-Commit frame from the supplicant, the access point replies with an new Auth-Commit frame with the anti-clogging token present. The client needs to reflect the token before the initial commit frame is processed. The cookie is created from the MAC addresses of both the supplicant and the access point. The authenticator alone may generate valid cookies, because she is in possession of the required random secret that is used to hash a concatenation of the MAC addresses of the supplicant and access point:

$$\text{anti\_clogging\_token} = \mathit{SHA256}(MAC_{ap}||MAC_{sta}||random)$$

By using tokens, the access point may throttle the processing of Auth-Commit frames based on the identity of the supplicant. The code responsible for processing anti-clogging tokens in the authenticator in `iwd 0.18` can be inspected in figure 20.

[20] `iwd`, accessed on 5th August 2019, `git://git.kernel.org/pub/scm/network/wireless/iwd.git`

[21] `ELL`, accessed on 5th August 2019, `https://01.org/ell`

```
static void sae_process_anti_clogging(struct sae_sm *sm, const uint8_t *ptr,
                                        size_t len)
{
        /*
         * IEEE 802.11-2016 - Section 12.4.6 Anti-clogging tokens
         *
         * It is suggested that an Anti-Clogging Token not exceed 256 octets
         */
        if (len > 256) {
                l_error("anti-clogging token size %zu too large, 256 max", len);
                return;
        }

        sm->token = l_memdup(ptr + 2, len - 2);
        sm->token_len = len - 2;
        sm->sync = 0;

        sae_send_commit(sm, true);
}
```

Figure 20: The function that parses anti clogging tokens in `iwd/src/sae.c v0.18`. [Int19]

An interesting question is if the frame length variable `len` can be 1 or 0, resulting in the expression `l_memdup(ptr + 2, len - 2);` which evaluates to `malloc(-1)` with a negative number -1. This invocation will result in an `abort()` panic as can be seen in figure 21.

```
LIB_EXPORT void *l_malloc(size_t size)
{
        if (likely(size)) {
                void *ptr;

                ptr = malloc(size);
                if (ptr)
                        return ptr;

                fprintf(stderr, "%s:%s(): failed to allocate %zd bytes\n",
                                        STRLOC, __func__, size);
                abort();
        }

        return NULL;
}
```

Figure 21: DoS vulnerability: Calling `malloc()` with negative values will abort the `iwd` daemon.

It turns out that a function stack leading to $len = 1$ is possible by forging a commit frame with a status code that indicates that it carries an anti-clogging token (status code $0x4c$ or WLAN_STATUS_ANTI_CLOGGING_TOKEN_REQ) but does deliberately have an empty payload.

```
Frame:
  0000  b0 97 20 02 00 00 00 00 18 01 02 00 00 00 00 01  .. ..............
  0010  50 99 20 02 00 00 00 00 00 1b 24 c3 03 00 01 00  P. ........$.....
  0020  4c 00 20                                         L.
```

Figure 22: Hexdump of a possible frame that triggers the DoS vulnerability in iwd 0.18. [Int19]

The frame must be a valid management frame with subtype authentication and the authentication methods must be SAE. The status code must be set to MMPDU_STATUS_CODE_ANTI_CLOGGING_TOKEN_REQ to properly reach the vulnerable sae_process_anti_clogging() function. The total length of the frame must be either 34 or 35 bytes, otherwise there won't be an subtraction of $len - 2$ that results in a negative number. If the frame has 35 bytes, the last byte may have any value. A possible frame that triggers the DoS vulnerability can be seen in figure 22.

Can the vulnerability be remotely triggered? When a vulnerable iwd supplicant tries to connect to an access point using SAE and the access point has anti-clogging tokens enabled, the supplicant initiates the WPA3-SAE handshake by sending an initial Auth-Commit frame. The access point replies with an Auth-Commit frame with the status set to MMPDU_STATUS_CODE_ANTI_CLOGGING_TOKEN_REQ. If this frame is a valid authentication frame using WPA3-SAE and has a length of 34 or 35 bytes, the supplicant will abort due to calling malloc with either -1 or -2. This will terminate iwd and the user will be unable to connect to this deliberately maliciously behaving access point.

A rouge access point could camouflage itself as another access point and thus an DoS attack may be launched against any BSSID. In the worst case, as soon as the DoS vulnerability was triggered, the iwd process aborts and does not restart.

Another plausible attacking scenario would be the injection of a malicious anti-clogging frame into an observed WPA3-SAE authentication handshake by spoofing the MAC address of the authenticator. This allows an remote attacker to crash any client trying to initiate a WPA3-SAE handshake, because Auth-Commit frames are not encrypted or protected and spoofing MAC addresses is trivial.

This security issue was fixed shortly after getting into contact with the maintainers of iwd 0.18 in commit 0241fe81dff67f4b134e01d10bd884e9509a9d6f[22] [Tsc19b].

[22] Commit that fixes the DoS, accessed on 5th August, https://git.kernel.org/pub/scm/network/wireless/iwd.git/commit/?id=0241fe81dff67f4b134e01d10bd884e9509a9d6f

6.3 Discussion

Research conducted during this thesis uncovered a DoS vulnerability in the handling of anti-clogging tokens (See section 6.2). This vulnerability was found during a manual code review and not with the hybrid fuzzing methodology applied in this thesis.

It is believed that a model based fuzzing approach is not the most promising strategy when bug hunting the WPA3-SAE handshake. A greybox and blackbox fuzzing strategy is more applicable to software that is heavy in parsing functionality, such as parsing information elements from beacon and probe request frames or handling EAPOL 4-way handshake messages.

A better fuzzing target would have been the entire `iwd` 802.11 supplicant software, because there are no previous fuzzing attempts observed and the supplicant has been open sourced only since 2016. Furthermore, `iwd` seems to be an viable candidate to replace `wpa_supplicant`. `hostapd` on the other side is continuously fuzzed as part of the distributed fuzzing project OSS-Fuzz [OSS19], which makes new fuzzing discoveries even less likely.

6.3.1 Practical Obstacles

Security research in the Wi-Fi ecosystem is nontrivial. There are many practical obstacles that dampen the development of an effective fuzzing framework. For example, a powerful fuzzer should be able to spoof its MAC address and receive all frames in reply to the spoofed source MAC address. This capability is required when stress testing the computational costly handling of Auth-Commit frames in WPA3-SAE access points.

To enable this functionality, a specific kernel module is required for each dedicated 802.11 network interface and firmware[23] that sends ACK management frames when receiving a frame with spoofed MAC address. However, developing kernel modules is a nontrivial task and requires significant understanding of Linux kernel internals and the mac80211 and nl80211 subsystems, as well as the specific firmware. For this reason, the fuzzing framework contributed by this thesis does not support generating 802.11 ACK frames in reply to frames with spoofed MAC addresses.

Furthermore, a remote fuzzer has to deal with various problems related to the unreliable radio medium: 802.11 frame loss and retransmission, detection of crashes of fuzzed access points, unreliable hardware or drivers and so on.

Another major practical issue is the detection of security implicative crashes in proprietary Wi-Fi hardware caused by fuzzed frames. Crash detection in the `Synology MR2200ac Router` investigated during this thesis was straightforward, because the

[23] Example of a kernel module extension which configures Atheros drivers to forward all frames from a specific MAC address range, accessed on 30th July 2019, `https://github.com/vanhoefm/ath_masker`

hostapd daemon process can be monitored over a wired SSH connection. However, not all proprietary Wi-Fi hardware provides shell access over an Ethernet connection. For example, many IoT devices or 802.11 repeaters have no such debugging functionality. For those devices, it is a nontrivial task to determine if and what frame caused a security critical crash.

6.3.2 Disadvantages of Remote Fuzzing

It is not entirely clear if remote fuzzing is the best strategy to uncover security vulnerabilities regarding WPA3-SAE. The only open source access point implementation that was targeted during this thesis was hostapd. The only WPA3 capable hardware tested during this thesis was the Synology MR2200ac Router using hostapd 2.7 internally.

Research conducted during this thesis has shown that WPA3 authentication is in most cases handled by a user space service process such as hostapd. However, in the first half of 2019, Cypress developers introduced SAE authentication offloading capabilities into the Linux kernel netlink interface nl80211, which allows wireless firmware to handle WPA3 authentication[24]. User space programs may provide the SAE password in a NL80211_ATTR_AUTH_DATA data structure via the nl80211 NL80211_CMD_CONNECT command, which requests to connect to a specified BSSID without separated authentication and association steps. The full release of the Cypress open source driver can be found online[25]. Cypress uses a broadcom wireless driver internally[26].

Therefore, it is possible that WPA3-SAE authentication is handled by the firmware itself, when the NL80211_FEATURE_SAE flag is advertised by the driver. As a consequence, an improved fuzzing policy compared to remote fuzzing would be to setup a Linux system with the whole collection of access point software and firmware under test on a single system. An immediate advantage would be the better monitoring capabilities and easier testing infrastructure compared to a remote fuzzing target. Additionally, it would be possible to monitor crashes in a straightforward way. If needed, physical 802.11 radios could be created between fully controllable machines.

6.3.3 Limitations of the Fuzzing Approach

Blackbox fuzzing with boofuzz as well as coverage-guided greybox fuzzing with fuzzing engines such as libFuzzer and AFL proved to be highly effective in detecting security vulnerabilities in many widely used software projects. For example, the Google initiated

[24] Mailing list discussion about SAE auth offload, accessed on 29th July 2019, https://patchwork.kernel.org/patch/10748075/

[25] Cypress Linux Wi-Fi Driver Release, accessed on 29th July 2019, https://community.cypress.com/docs/DOC-17441

[26] Broadcom wireless driver https://github.com/torvalds/linux/tree/master/drivers/net/wireless/broadcom/brcm80211

cloud fuzzing project OSS-Fuzz uncovered around 14.000 bugs in over 200 open source projects as of August 2019 [OSS19].

The SAE handshake however consists of only two frames with a total of five possible fields in the Auth-Commit management body (group id, scalar, element, optional anti-clogging token, optional password identifier) and only two possible fields in the Auth-Confirm frame (send confirm number, confirm token). Therefore, the parsing of those two authentication frames has limited complexity and thus the likelihood of programming mistakes in the parsing code is similarly slim. Hence, a solely fuzzing based approach as conducted in this thesis covers a fraction of all existing vulnerability classes.

As the recent security survey of the WPA3-SAE handshake of Vanhoef et al. has shown [VR19], all of their discovered vulnerabilities were either logical security vulnerabilities, downgrade attacks, side channel attacks or Denial of Service attacks. They did not manage to find classical programming mistakes that lead to security implications. However, it is extremely unlikely that such logical vulnerabilities are triggered with a blackbox or greybox fuzzing based method as conducted during this thesis.

6.3.4 Symbolic Execution instead of Fuzzing

Using white box fuzzing approaches such as symbolic execution instead of greybox and blackbox fuzzing is a possibly lucrative strategy for future work. Symbolic execution guarantees full code coverage by generating symbolic inputs instead of concrete ones.

Execution forks as soon as a new code branch is encountered and state conditions are recorded on each new conditional branch. Whenever a concrete test case needs to be generated, a satisfiability modulo theories (SMT) resolver is applied [CDE+08]. However, symbolic execution is expensive in terms of computational resources and not practically feasible in implementations that make use of extensive cryptographic libraries due to an explosion of possible branches within cryptographic primitives [VP18b].

A whitebox fuzzing method was used in recent research to fuzz the 4-way handshake in open source 802.11 software such as `hostap` [VP18b]. The methodology has proven to be promising, motivating the application of similar endeavors in the WPA3-SAE handshake as proposal for future work.

In this thesis, even though being aware of the advantages of symbolic execution, it was decided against it for the following reasons:

1. The output of cryptographic primitives would have to be treated as a fresh symbol, because symbolic execution of algorithms such as AES or HMAC have a computationally large impact when being subjected to a SMT solver [VP18b].
2. The relevant functionality of WPA3-SAE must be isolated from the huge codebase

of hostapd in order to make it suitable for symbolic execution. For example, the functions sae_parse_commit() and sae_check_confirm() would need to be isolated in the file hostapd-2.8/src/common/sae.c. This is problematic, because the isolation of functionality creates new vulnerabilities while existing flaws are destroyed.

6.4 Conclusion

Writing an efficient fuzzer that targets the WPA3-SAE authentication handshake in the wild is a difficult assignment, because there exists almost no hardware that supports the WPA3-SAE handshake at the time of writing this thesis (August 2019).

Instead a hybrid approach was followed. One method was the in-process, coverage-guided greybox fuzzing of the open source Wi-Fi supplicant iwd and access point implementation hostapd with the fuzzing engine libFuzzer.

Another fuzzing strategy was a remote, over-the-radio blackbox fuzzing of the Synology MR2200ac Router, one of the rare devices that already supports the new WPA3 certification. Remote fuzzing was conducted with a fuzzing framework named *Dragonfuzz* that was developed during this thesis. This blackbox fuzzing engine targets specific states in the WPA3-SAE handshake which were inferred from a finite state machine model derived in section 5.4.

As a concluding statement, the author conjectures that a fuzzing based approach using modern, powerful greybox fuzzing engines such as AFL or libFuzzer is only meaningful, if the target of evaluation is rich in parsing functionality. This is not necessarily the case with WPA-SAE implementations, where parsing is limited to a few fields of static size. The room for logical flaws such as timing attacks or cryptographic implementation mistakes is much larger, as recent research has proven [VR19].

A manual security audit that checks for logical vulnerabilities is probably more successful in uncovering security vulnerabilities compared to a automated fuzzing based methodology. However, such a manual review process requires extensive experience from the auditor in various areas of computer security research in order to yield potential results.

Nevertheless, the research conducted in this thesis yielded a harmful DoS vulnerability in the 802.11 supplicant software iwd and thus justifies the chosen methodology.

List of Figures

List of Tables

References

[Adr+15] David Adrian et al. “Imperfect forward secrecy: How Diffie-Hellman fails in practice”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM. 2015, pp. 5–17.

[All19a] WiFi Alliance. *WPA3 Standard.* `https://www.wi-fi.org/download.php?file=/sites/default/files/private/WPA3_Specification_v1.0.pdf`. WiFi Alliance, 2018, [Online; accessed on 21th July 2019].

[All19b] WiFi Alliance. *WiFi Allicance WPA3-Personal Product Finder.* `https://www.wi-fi.org/product-finder-results?sort_by=certified&sort_order=desc&categories=4,6&capabilities=16`. www.wi-fi.org, 2019, [Online; accessed on 29th July 2019].

[AO16] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2016.

[AVM19] AVM. *Avm security news.* `https://avm.de/service/aktuelle-sicherheitshinweise/`. AVM, 2019, [Online; accessed on 29th July 2019].

[Bel19] Fabrice Bellard. *QEMU.* `https://www.qemu.org/`. QEMU.org, 2019, [Online; accessed on 21th July 2019].

[Beu+15] Benjamin Beurdouche et al. “A messy state of the union: Taming the composite state machines of TLS”. In: *Security and Privacy (SP), 2015 IEEE Symposium on.* IEEE. 2015, pp. 535–552.

[BM19] S. Bellovin and M. Merritt. *U.S. Patent 5,241,599. - Cryptographic protocol for secure communications.* `https://patents.google.com/patent/US5241599A/en`. Google Patents, 1991, [Online; accessed on 21th June 2019].

[Böh+17] Marcel Böhme et al. “Directed greybox fuzzing”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM. 2017, pp. 2329–2344.

[Boo19] Boofuzz. *Boofuzz fuzzing framework, Sulley successor.* `https://boofuzz.readthedocs.io/en/latest/`. Github.com, 2019, [Online; accessed on 29th July 2019].

[BPR17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-based greybox fuzzing as markov chain”. In: *IEEE Transactions on Software Engineering* 45.5 (2017), pp. 489–506.

[BSI19] BSI. “Empfehlungen und Schlüssellängen, Tabelle 3.1”. In: *Technische Richtlinie TR-02102-1, Bundesamt für Sicherheit in der Informationstechnik (BSI), Stand 22. Februar 2019* (2019), p. 28.

[BT08] Laurent Butti and Julien Tinnes. "Discovering and exploiting 802.11 wireless driver vulnerabilities". In: *Journal in Computer Virology* 4.1 (2008), pp. 25–37.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[CH14] Dylan Clarke and Feng Hao. "Cryptanalysis of the dragonfly key exchange protocol". In: *IET Information Security* 8.6 (2014), pp. 283–289.

[Cha+19] Oliver Chang et al. *OSS-Fuzz: Five months later, and rewarding projects.* `https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html`. Google, 2017, [Online; accessed on 30th August 2019].

[Che+18] Jiongyi Chen et al. "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing." In: *NDSS*. 2018.

[Clu19] Clusterfuzz contributors. *Coverage-guided vs blackbox fuzzing.* `https://google.github.io/clusterfuzz/`. Github.com, 2019, [Online; accessed on 3th August 2019].

[DP15] Joeri De Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations." In: *USENIX Security Symposium*. 2015, pp. 193–206.

[Gas05] Matthew Gast. *802.11 Wireless Networks: The Definitive Guide, 2nd Edition by Matthew S. Gast.* Chapter 4. 802.11 Framing in Detail, Section: Management Frames. O'Reilly Media, Inc., 2005.

[Har08] D. Harkins. "Simultaneous Authentication of Equals: A Secure, Password-Based Key Exchange for Mesh Networks". In: *2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008)*. Aug. 2008, pp. 839–844. DOI: `10.1109/SENSORCOMM.2008.131`.

[Har15] D. Harkins. *Dragonfly Key Exchange*. RFC 7664. RFC Editor, Nov. 2015, pp. 1–18. URL: `https://www.rfc-editor.org/rfc/rfc7664.txt`.

[Har19a] D. Harkins. *Secure Password Ciphersuites for Transport Layer Security (TLS)*. RFC 8492. RFC Editor, Feb. 2019, pp. 1–40. URL: `https://www.rfc-editor.org/rfc/rfc8492.txt`.

[Har19b] Dan Harkins. *Dragonfly: A PAKE Scheme.* `https://datatracker.ietf.org/meeting/83/materials/slides-83-cfrg-0`. IETF 83, 2012, [Online; accessed on 6th July 2019].

[Har19c] Dan Harkins. *Reference Implementation of WPA3-SAE.* `https://sourceforge.net/p/authsae/wiki/Home`. sourceforge.net, 2014, [Online; accessed on 21th June 2019].

[Har19d] Dan Harkins. *DCN 387, Group TGm, Addressing some SAE comments.* `https://mentor.ieee.org/802.11/dcn/19/11-19-0387-02-000m-addressing-some-sae-comments.docx`. IEEE, March 2019, [Online; accessed on 16th June 2019].

[Hie+10] Guido R Hiertz et al. "IEEE 802.11s: the WLAN mesh standard". In: *IEEE Wireless Communications* 17.1 (2010), pp. 104–111.

[HR10] Feng Hao and Peter Ryan. "J-PAKE: authenticated key exchange without PKI". In: *Transactions on computational science XI.* Springer, 2010, pp. 192–206.

[htt19] `https://stackoverflow.com/users/4975822/artm`. *Architecture of modern Linux kernels.* `https://stackoverflow.com/questions/21456235/how-nl80211-library-cfg80211-work`. stackoverflow.com, Octobre 2015, [Online; accessed on 28th August 2019].

[IEE16] IEEE. *IEEE Std 802.11. 2016. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec.* IEEE, 2016.

[Int19] Intel. *iwd - iNet wireless daemon.* Version 0.18. May 11, 2019. URL: `https://git.kernel.org/pub/scm/network/wireless/iwd.git/`.

[Jab19] D. Jablon. *U.S. Patent 6,226,383. - Cryptographic methods for remote authentication.* `https://patents.google.com/patent/US6226383`. Google Patents, 1997, [Online; accessed on 21th July 2019].

[Jou19] Jouni Malinen and contributors. *Hostapd.* Version 2.8. Apr. 21, 2019. URL: `http://w1.fi/hostapd/`.

[Kav19] Sumanth Kavuri. *Protocol Stack in Wi-Fi Chipsets.* `http://80211notes.blogspot.com/2014/08/protocol-stack-in-wi-fi-chipsets.html`. 80211notes.blogspot.com, 2014, [Online; accessed on 16th July 2019].

[KK07] Sylvester Keil and Clemens Kolbitsch. "Stateful fuzzing of wireless device drivers in an emulated environment". In: *Black Hat Japan* (2007).

[LK14] Yehuda Lindell and Jonathan Katz. *Introduction to modern cryptography.* Chapman and Hall/CRC, 2014.

[LŠ15] Jean Lancrenon and Marjan Škrobot. "On the Provable Security of the Dragonfly protocol". In: *International Information Security Conference.* Springer. 2015, pp. 244–261.

[Man+18] Valentin JM Manes et al. "Fuzzing: Art, science, and engineering". In: *arXiv preprint arXiv:1812.00140* (2018).

[MFS90] Barton P Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[MN08] Manuel Mendonça and Nuno Neves. "Fuzzing wi-fi drivers to locate security vulnerabilities". In: *Dependable Computing Conference, 2008. EDCC 2008. Seventh European.* IEEE. 2008, pp. 110–119.

[Nik19] Nikolai Tschacher. *Dragonfuzz Framework.* Version 1.0. `https://gitlab.com/NikolaiT/dragonfuzz`, `https://github.com/NikolaiT/dragonfuzz/`, `https://github.com/NikolaiT/fuzz_sae_hostap` commit 0101c59. 2019, [Online; accessed on 2nd September 2019].

[OSS19] OSS-fuzz. *Continuous Fuzzing for Open Source Software - OSS-fuzz.* `https://github.com/google/oss-fuzz/`. Github.com, 2019, [Online; accessed on 3th August 2019].

[Per19] Trevor Perrin. *TLS Review of Dragonfly PAKE.* `https://mailarchive.ietf.org/arch/msg/tls/A_SfHI4BsdAi4miklBs3TvUbu-Y`. Windows Phone Central, December 2013, [Online; accessed on 23th June 2019].

[PP09] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners.* Springer Science & Business Media, 2009.

[Tsc19a] Nikolai Tschacher. *dragonfuzz.c, commit 42bb1e34.* `https://gitlab.com/NikolaiT/dragonfuzz`. Github.com, 2019, [Online; accessed on 21th July 2019].

[Tsc19b] Nikolai Tschacher. *Iwd v0.18 Denial of Service.* `https://git.kernel.org/pub/scm/network/wireless/iwd.git/tree/src/sae.c`, Commit: 0241fe81dff67f4b134e01d10bd884e9509a9d6f. git.kernel.org, 2019, [Online; accessed on 29th July 2019].

[Tsc19c] Nikolai Tschacher. *dragonfuzz.py, commit d19be24.* `https://github.com/NikolaiT/dragonfuzz/`. Github.com, July 2019, [Online; accessed on 21th July 2019].

[Van19] Maty Vanhoef. *Dragondrain.* `https://github.com/vanhoefm/dragondrain-and-time/`. Github.com, 2019, [Online; accessed on 21th July 2019].

[VP17] Mathy Vanhoef and Frank Piessens. "Key reinstallation attacks: Forcing nonce reuse in WPA2". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM. 2017, pp. 1313–1328.

[VP18a] Mathy Vanhoef and Frank Piessens. "Release the Kraken: New KRACKs in the 802.11 Standard". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* ACM. 2018, pp. 299–314.

[VP18b] Mathy Vanhoef and Frank Piessens. "Symbolic execution of security protocol implementations: handling cryptographic primitives". In: *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18).* 2018.

[VR19] Mathy Vanhoef and Eyal Ronen. "Dragonblood: A Security Analysis of WPA3's SAE Handshake." In: *IACR Cryptology ePrint Archive* 2019 (2019, [Online; accessed on 30th August 2019]). `https://eprint.iacr.org/2019/383`, p. 383.

[VSP17] Mathy Vanhoef, Domien Schepers, and Frank Piessens. "Discovering logical vulnerabilities in the Wi-Fi handshake using model-based testing". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM. 2017, pp. 360–371.

[wir19] wireless.wiki.kernel.org. *WiFi Glossary*. `https://wireless.wiki.kernel.org/en/developers/documentation/glossary`. 2019, [Online; accessed on 30th July 2019].

[wla19] wlan1nde.wordpress.com. *WPA3, improving your wlan security*. `https://wlan1nde.wordpress.com/2018/09/14/wpa3-improving-your-wlan-security/`. wlan1nde.wordpress.com, 2018, [Online; accessed on 21th July 2019].

[Zel+19a] Andreas Zeller et al. "Generating Software Tests". In: *Generating Software Tests*. Retrieved 2019-05-21 20:25:44+02:00. Saarland University, 2019. URL: `https://www.fuzzingbook.org/html/00_Table_of_Contents.html`.

[Zel+19b] Andreas Zeller et al. "Greybox Fuzzing". In: *Generating Software Tests*. Retrieved 2019-05-19 14:42:27+02:00. Saarland University, 2019. URL: `https://www.fuzzingbook.org/html/GreyboxFuzzer.html`.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den September 10, 2019 ....