SAECRED: A State-Aware, Over-the-Air Protocol Testing Approach for Discovering Parsing Bugs in SAE Handshake Implementations of COTS Wi-Fi Access Points

Muhammad Daniyal Pirwani Dar[§], Rob Lorch[†], Aliakbar Sadeghi[§], Vincenzo Sorcigli[§], Héloïse Gollier[‡], Cesare Tinelli[†], Mathy Vanhoef[‡], and Omar Chowdhury[§]

[§]*Stony Brook University* [†]*The University of Iowa* [‡]*DistriNet, KU Leuven*

[§]*{mdar, alisadeghi, omar}@cs.stonybrook.edu* [§]*vincenzo.sorcigli@stonybrook.edu*

[†]*{robert-lorch, cesare-tinelli}@uiowa.edu* [‡]*{heloise.gollier, mathy.vanhoef}@kuleuven.be*

***Abstract*—WPA3-Personal introduced the *stateful* Simultaneous Authentication of Equals (SAE) handshake protocol to achieve forward secrecy and resistance to passphrase guessing attacks during Wi-Fi connection bootstrapping, guarantees that are lacking in WPA2-Personal. However, the initial design of WPA3-Personal with SAE was susceptible to connection downgrade and denial-of-service (DoS) attacks. The current, enhanced version introduces mechanisms to mitigate these vulnerabilities. Enabling these security-enhancing mechanisms, however, results in a variable-structured, context-sensitive packet format that can be challenging to parse and interpret correctly. Misparsing SAE handshake packets can negatively impact Wi-Fi protocol security. To uncover SAE handshake packet misparsing in commercial-off-the-shelf (COTS) Wi-Fi access points (APs), we present SAECRED, a packet-structure-guided, SAE-state-aware black-box fuzzer. SAECRED reduces the underlying problem of misparsing discovery to a two-dimensional search problem, where the dimensions are the packet structure and the underlying SAE protocol state. It solves this search problem by combining Iterative Deepening Search (IDS) with a context-sensitive grammar-based fuzzing approach, where the latter relies on a Syntax-Guided Synthesis (SyGuS) solver. SAECRED's effectiveness is demonstrated by evaluating it on 6 COTS APs and the widely used open-source `hostapd`. Our evaluation discovered several instances of 4 classes of bugs. Bugs in two of these classes violate the two fundamental guarantees SAE expects to achieve (*i.e.*, resistance to downgrade and DoS attacks). We reported our findings to the relevant stakeholders, which resulted in patches and security advisories.**

1. Introduction

The ubiquity of Wi-Fi and its over-the-air (OTA) communication makes it a lucrative target for local adversaries. Wi-Fi security, and especially its connection bootstrapping protocol's security, has been the focal point of numerous recent efforts [1], [2], [3], [4]. The Wi-Fi connection bootstrapping protocol in the WPA2-Personal Mode[1] lacks forward secrecy and is also susceptible to brute-force attacks on the secret passphrase shared between the Wi-Fi Access Point (AP) and the user (a.k.a., *supplicant*) [5]. To address these security concerns of WPA2, the IEEE 802.11 standard went through multiple revisions [6], [7] before settling on the WPA3 standard. The WPA3 standard introduces a *stateful* password-authenticated key exchange (PAKE) protocol called the Simultaneous Authentication of Equals (SAE). The SAE protocol enjoys both forward secrecy and resistance to offline dictionary attacks on the credentials. *At a high level, this paper focuses on checking the correctness of SAE implementations in commercial-off-the-shelf (COTS) APs.*

Tool	Mode	Grammar	Mutation	State	Bugs
GREYHOUND	◐	✗	**B/P**	✓	●
StateAFL	◐	✗	**B/P**	✓	●
Braktooth	◐	✗	**B/P**	✓	●
TCP-Fuzz	◐	✗	**P**	✓	◐
Tyr	◐	✗	**P**	✓	◐
SGFuzz	◐	✗	**B**	✓	●
Nautilus	◐	✓	**G**	✗	○
SNPSFuzzer	◐	✗	**B/P**	✓	○
StateFuzz	◐	✗	**B**	✓	○
EvoGFuzz	◐	✓	**G**	✗	○
Superion	◐	✓	**G**	✗	○
Owfuzz	●	✗	**B**	✗	○
AFLNet	●	✗	**B/P**	✓	○
PULSAR	●	✗	**B/P**	✓	●
FuzzPD	●	✗	**P**	✓	◐
DPIFuzz	●	✗	**B/P**	✗	◐
BLEEM	●	✗	**B/P**	✓	●
AFLNetLegion	●	✗	**B/P**	✓	○
L2Fuzz	●	✗	**B**	✓	○
ATFuzz	●	✓	**G**	✗	●
DIKEUE	●	NA	NA	✓	◐
TLSAttacker	●	✗	**B/P**	✓	●
SAECRED	●	✓	**G/P**	✓	●

TABLE 1: Comprehensive fuzzer capability matrix. Mode: ○: white-box, ◐: gray-box, ●: black-box. Mutation: **B**: bit-level, **P**: packet-level, **G**: grammar-based, **B/P**: bit+packet, **G/P**: grammar+packet. Capability: ○: crashes, ◐: semantic bugs, ●: crashes+semantic bugs.

Despite its secure-by-design motto, the security of the

1. Unless explicitly mentioned, whenever we refer to WPA2 or WPA3, we mean the WPA2-Personal and WPA3-Personal modes, respectively.

SAE handshake protocol has been under constant scrutiny due to design flaws and implementation choices. Concretely, the initial design and implementation in the widely used `hostapd` library [8] were shown to be susceptible to connection downgrade and DoS attacks [2], [9]. These findings led to the current enhanced version of the SAE standard, which introduces mechanisms (*e.g.*, explicitly tracking rejected elliptic curve groups, introducing anti-clogging tokens) for mitigating these attacks. To enable these security-enhancing mechanisms, however, the SAE handshake packet structure has become more complex. These packets have variable lengths, with inter-field dependencies and cryptographic contexts. Precisely capturing this structure requires a context-sensitive grammar with calculated fields. As a consequence, parsing the packet format correctly and then faithfully updating internal and protocol states can be a challenging and error-prone task for AP implementations. Misparsing SAE handshake packets can lead to several undesirable consequences: erroneous protocol state transitions, memory safety issues (*e.g.*, crashes, leakage of sensitive information), undefined behavior, and unsafe or erroneous internal state updates (*e.g.*, updating an internal variable with incorrect values, impacting future protocol behavior). These undesired consequences can be exploited by adversaries to launch attacks ranging from DoS to connection downgrades. *To mitigate these issues we designed, developed, and experimentally evaluated* SAECRED*, a packet-structure-guided, SAE-state-aware black-box over-the-air testing approach for uncovering parsing bugs in COTS APs.*

Due to the SAE protocol's statefulness, discovering parsing bugs is challenging. In particular, triggering a parsing bug may require not only generating the concrete SAE packet (potentially nonconformant with the SAE packet format) that triggers the bug but also injecting it only when the AP is in a SAE state that does trigger the bug. Reaching the bug-triggering SAE state may require sending a long sequence of SAE packets, which induces a large input universe to sample from. Since we consider a black-box setting, justified by the lack of access to proprietary AP firmware, we do not have information about the internal SAE protocol state of the AP under test. Further, we do not have faithful coverage information (*e.g.*, code coverage) to guide the input generation part of testing, where each input is a sequence of SAE protocol packets. A popular choice is using proxy coverage measures, such as the execution time of a test case, to approximate code coverage. Unfortunately, such proxy measures tend to guide the testing approach towards inputs that trigger timeouts rather than parsing bugs.

One key advantage of our testing setup is that one can easily test a COTS AP with SAECRED with little to no configuration or programming efforts. The setup also requires an SAE-aware driver that can not only inject the test inputs OTA to the AP under test but also decode and classify the AP responses as expected or unexpected. The driver has to support such intricacies of the SAE protocol as cryptographic operations, contextual behavior (*e.g.*, handling anti-clogging tokens), timeouts, and retransmissions. Finally, as with testing of stateful wireless protocols, executing each test case may take a long time as the driver has to account for timeouts and retransmissions, and also reset the AP to a known initial state before injecting the next test case. Faithfully resetting may require power cycling the AP.

Ignoring the SAE-specific driver (which is essential for testing APs), Table 1 summarizes the existing gray-box and black-box testing approaches, including those specifically targeting stateful network protocols. These fuzzers are unsuitable for our testing setup because they suffer from one or more of the following limitations: (1) they require access to source code or the firmware binary [10], [11], [12], [13], [14]; (2) they do not consider malformed packets [15], [16], [17]; (3) they lack support for stateful protocol behavior [18], [13], [19], [20], [21]; (4) they cannot identify undesired parsing behavior that does not induce a crash [13], [14], [22], [23], [20], [17], [24], [25]; (5) they consider inputs of certain shapes only [26], [27]; (6) they cannot handle packet formats that can be expressed only with context-sensitive grammars [13], [15], [11], [18], [28], [29]; (7) they rely on test cases having a short execution time.[10], [18], [15], [11]. A more detailed breakdown of the limitations of these works in the WPA3-SAE context can be found in Section 10. One *key insight* that enables SAECRED to address these limitations is that discovering parsing bugs in SAE implementations can be reduced to a search problem over two dimension: the packet structure and the underlying SAE protocol state space. SAECRED solves this two-dimensional search problem by combining IDS with a grammar-based fuzzing technique that uses context-sensitive grammars.

Generally speaking, SAECRED follows an iterative approach. Starting from some *fixed* seed inputs, at each iteration, it generates new test inputs — to be executed on the AP under test — by mutating specific inputs from the previous iteration, chosen based on some scoring criteria. Internally, SAECRED can be viewed as running an IDS along with grammar-based fuzzing. Concretely, for each input we keep track of the inferred SAE protocol state, the path (*i.e.*, a sequence of concrete SAE packets) that got us to that state, and the packet format to generate the next concrete SAE packet. The packet format is specified by a context-sensitive grammar, which can be viewed as an extension of a Backus-Naur form (BNF) grammar with constraints on some of its syntactic categories (*e.g.*, the length of a certain packet field f_1 must be equal to the value of another packet field f_2). Mutations in SAECRED take the form of mutating the grammar instead of concrete packets. The mutations can result in (1) the structural modification of the packet format or (2) the modification of some data-dependent constraints in the grammar. After the grammar is mutated, SAECRED uses a syntax-guided program synthesizer (SyGuS [30]) to generate a concrete SAE packet that conforms to the mutated grammar. Mutating the grammar, rather than mutating concrete packets, has two key advantages: (1) it allows SAECRED to explore subtle changes in both the packet structure and the data-dependent constraints, which is difficult to achieve by mutating concrete packets; (2) if an input generate from a grammar causes a parsing bug in the AP under test, the grammar can be used as symbolic evidence of the bug's root

cause. Another notable aspect of SAECRED is that it tries to make progress in the testing campaign by partitioning the generated inputs in the previous iteration based on their inferred SAE protocol states. This categorization of inputs from the previous iteration enables SAECRED to implement prioritization strategies in which inputs that take the AP to dangerous states are given priority over benign inputs. This allows SAECRED to avoid getting stuck in the initial state.

SAECRED is implemented in a combination of OCaml and Python and uses cvc5SY as its SyGuS solver [31], [32]. We evaluated the effectiveness of SAECRED on 6 COTS APs and the widely used open-source `hostapd` library. We observed that SAECRED discovered instances of 4 classes of bugs, two of which violate the two fundamental guarantees SAE is designed to provide (*i.e.*, resistance to downgrade and DoS attacks). It also achieves a substantially high code coverage on the open-source `hostapd` library compared to some baselines. Notable among its findings is the bug that causes the misparsing of an optional container, which allows a Dolev-Yao-style network attacker to downgrade the connection to a weaker security level. We reported all our findings to the vendors. At the time of writing, some of the vendors have already released patches and security advisories, whereas others are still investigating them.

In summary, our work makes the following *technical contributions*:

1) SAECRED reduces the problem of discovering parsing bugs in SAE implementations to a two-dimensional search problem, which it solves by combining IDS with a context-sensitive-grammar-based fuzzing approach.
2) SAECRED contains an SAE-aware driver that can inject test inputs OTA to the AP under test, and decode and classify the AP responses as expected or unexpected. The driver can enable other testing algorithms and is of independent interest.
3) SAECRED discovered 4 classes of bugs in 6 COTS APs and the open-source `hostapd` library.
4) The entire SAECRED system is open source.[2]

2. Background

The SAE handshake was added to the 802.11 standard in 2011 to secure mesh networks [33]. A close variant of this handshake, called Dragonfly, was later standardized in RFC 7764 [34]. It became mandatory in 2018 for home networks in the new WPA3 specification. In contrast to WPA2, SAE offers forward secrecy and prevents dictionary attacks. Unfortunately, there are various side-channel leaks in the hash-to-curve algorithm used to derive keys [2]. To mitigate these vulnerabilities, a new hash-to-element algorithm was introduced in the 2020 update to the IEEE 802.11 standard.

The SAE handshake has commit and confirm phases. The commit phase negotiates a fresh key, and the confirm phase checks that both parties agreed on the same key. The format of commit and confirm frames[3] is shown in Figure 2a and 2b, respectively. All commit frames, excluding those with an error status, have a scalar and an element that are used to negotiate a fresh key. The lengths of these fields depend on the finite cyclic group being used, where WPA3 mandates that every implementation supports at least group 19, which is a 256-bit elliptic curve group.

2.1. SAE Handshake

The SAE handshake for infrastructure Wi-Fi networks, where a central AP manages all clients, consists of three states: (1) `Nothing`, (2) `Confirmed`, and (3) `Accepted`. The standard-specified state machine also supports mesh networks and consists of four states. The client-server architecture state machine, used in infrastructure networks, is simpler, with a straightforward protocol flow: clients and APs start in the `Nothing` state and make their way toward the `Accepted` state. Figure 1 shows a successful handshake flow: clients initiate the handshake through sending commit frames that consist of (1) a password-element (`PWE`) algorithm (hunting-and-pecking or hash-to-element `H2E`), (2) an ECC group, and (3) two cryptographically generated values (scalar and element) based on the chosen ECC group. If the `PWE` algorithm is `H2E`, the client has the option to include Information Elements (IEs or containers)[4] at the end of the frame, as shown in Figure 2a.

If the AP supports the client-chosen `PWE` algorithm and ECC group, it will respond with its own commit frame with two cryptographically generated values. If the AP does not support the chosen ECC group, it will respond with an `UNSUPPORTED_CYCLIC_GROUP` error. If the cryptographic values conform to the standard-defined range and lie on the ECC curve, both parties transition to the `Confirmed` state.

Both parties now enter the confirm phase where the goal is to *confirm* that both parties generated the same Pairwise Master Key (`PMK`). The `PMK` will be used in deriving the session key (`PTK`); therefore, both sides must confirm that they did indeed derive the same key. The parties generate confirm hashes through an `HMAC256`-based key derivation function (`KDF`) and authenticate each other through the confirm phase. If both parties successfully match the *received* confirm hashes with their internally generated *expected* confirm hashes, they transition to the `Accepted` state.

The SAE handshake is now complete; the client will now signal the successful termination of the handshake through an `ASSOCIATION_REQUEST` to which the AP responds with an `ASSOCIATION_RESPONSE` and begins the four-way handshake for `PTK` generation.

2.2. SAE Hash-To-Element

Hash-To-Element. In the hash-to-element update of SAE, which was meant to prevent side-channel flaws [2], the crux of the handshake still consists of commit and confirm

2. Available at https://github.com/izdar/SAECRED

3. We use frames and packets interchangeably to refer to Wi-Fi frames.

4. Information Elements are referred to as containers in the 802.11 2020 Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification [7]; we will refer to them interchangeably.

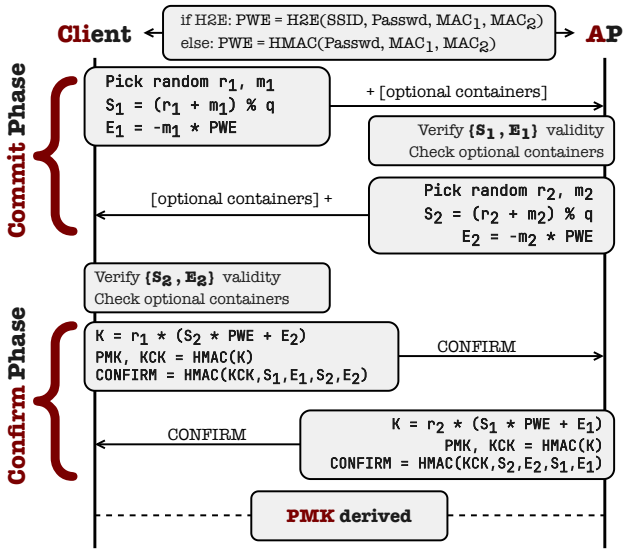


Figure 1: Complete breakdown of the SAE handshake with support for hash-to-element (H2E).

phases. This update introduced the use of optional IEs, also called containers, to offer added security guarantees. The format of a container embedded in a commit frame is shown in Figure 2a, where after the ELEMENT ends, the containers begin. Each container contains a tag, length, extension ID, and value, where the extension ID defines the type of container. The major addition to the handshake is the inclusion of three containers: (1) Password Identifier, (2) Rejected Groups, and (3) AC Token.

Rejected Groups. In SAE, the client proposes which cryptographic cyclic group to use. The AP can reject unsupported groups, after which the client can propose a different group. This procedure has been shown to be vulnerable to downgrade attacks [2]. To prevent them, the commit frame was extended to contain a list of cyclic groups rejected by the sender. This defense is only enabled when using the newer and more secure hash-to-element algorithm to derive keys. To make parsing feasible, the rejected groups are included as an optional IE at the end of the commit frame.

The inclusion of the optional rejected groups container allows for a check on the AP side to detect downgrade attacks. In addition to checking the container, the KDF on both the client and AP sides must generate keys using the list of rejected groups as a parameter. This guarantees downgrade protection, given a Dolev-Yao [35] adversary capable of manipulating packets and removing rejected groups from commit frames, since both parties will not derive the same pairwise master key (PMK) when a downgrade attack is attempted. All combined, the added parameter to the KDF ensures that active adversaries cannot successfully downgrade connections through trivial packet manipulation.

Password Identifiers. SAE successfully introduced forward secrecy and made brute force attacks infeasible. However, plain SAE does not support multiple passwords in a single network because both the AP and client must *commit* to a specific password when executing the SAE handshake. To support multiple passwords in a single network, the commit frame was later extended to include the use of an optional password identifier container. On receipt of a commit frame, the AP can then use the password identifier to also *commit* to the same password. When using this extension, in addition to computing intermediate keys based on the SSID, password, and client/AP MAC addresses, both parties also use the password identifier as an extra parameter to intermediate key generation as a defense in depth. APs can then be configured to support a set of password identifiers, and clients must include one of the configured password identifiers in their commit frames. In case there is a mismatch, APs must terminate the handshake. Password identifiers introduce complexities, as their exact implementation is up to the APs.

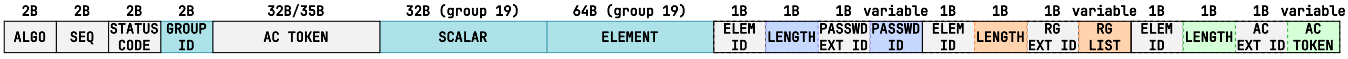
Anti-Clogging (AC) token. In an attempt to reduce DoS risks, the access point can require clients to reflect an AC token. Originally, this optional AC token precedes the scalar and element (see Figure 2a). This makes parsing commit frames nontrivial: a receiver infers whether an AC token is present based on the frame's length. Moreover, an AC token's length is not fixed but chosen by the AP, which further complicates frame parsing. The designers recognized this complexity [36], and in the hash-to-element update, encapsulated the AC token inside an AC token container. This solves the variable length problem, as the length of the element is included in the container, making parsing AC tokens simpler.

3. Problem, Threat Model, and Challenges

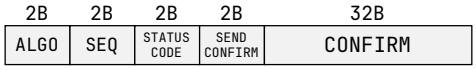
We now present a motivating example for SAECRED's approaches for finding parsing bugs. We then present the formal definition of the underlying problem, the threat model, and the technical challenges one encounters when designing a SAECRED-like testing approach.

Motivating Example. To motivate SAECRED and its underlying design, we discuss the following security-critical bug SAECRED discovered. The 802.11 standard defines the use of a rejected groups container in case the AP responds to a commit message with an UNSUPPORTED_CYCLIC_GROUP error code. If an AP sends this commit error message, the client must append the rejected group as part of the rejected groups container. This protects against ECDH group downgrade attacks, as adversaries spoofing AP responses and sending the UNSUPPORTED_CYCLIC_GROUP commit errors would not be able to downgrade the chosen group, as the client will append this group to their next commit message. The AP can then verify whether the rejected group is supported, and if so, it will detect the downgrade attack.

SAECRED generated a test case that mutated the length field of the rejected groups container. It then ran that test case against hostapd and reported that it was expecting a commit error, but received a valid commit message from the AP. We discuss the implications and details of this attack in Section 9.1, but the discovery of this attack in a widely



(a) SAE commit frame: Data dependent fields are color-coded together indicating the parsing context.



(b) SAE confirm frame.

Figure 2: The SAE handshake packet formats.

used and tested library motivates the need for approaches like SAECRED for finding parsing bugs in Wi-Fi APs.

Problem Definition. The SAE handshake protocol can be abstractly represented as a state transition system of the form: $M \triangleq \langle \mathcal{S}, s_0, \mathcal{L}, \Pi, \mathcal{R} \rangle$, in which $\mathcal{S}$ is the set of states, $s_0 \in \mathcal{S}$ is the initial state, $\mathcal{L}$ is the universe of SAE packets, Π is a function that maps a state to a set of expected packets in that state (*i.e.*, $\Pi : \mathcal{S} \rightarrow \mathcal{L}$), and $\mathcal{R}$ is the transition relation where $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$. Concretely, a *valid transition* $t \triangleq s_i \xrightarrow{p} s_j$ in M (*i.e.*, $t \in \mathcal{R}$) satisfies the following conditions: $s_i, s_j \in \mathcal{S}$ and $p \in \Pi(s_i)$. A protocol execution trace $\sigma \triangleq \langle s_0 \xrightarrow{p_0} s_1 \xrightarrow{p_1} s_2 \ldots s_{(n-1)} \xrightarrow{p_{(n-1)}} s_n \rangle$ is *considered legitimate* in M if and only if every transition $t_i \triangleq s_{(i)} \xrightarrow{p_{(n-1)}} s_{(i+1)}$ in σ satisfies the condition $t_i \in \mathcal{R}$ where $0 \leq i \leq (n-1)$. Given the SAE protocol model M and an AP implementation I of M, the discovery of parsing bugs in I is to find a set of packet sequences which induces executions Σ in I such that each $\sigma \in \Sigma$ is *considered legitimate* according to I but not in M.

Threat Model. In the SAE context, we consider a Dolev-Yao-style [35] network adversary capable of passively observing and actively engaging in Wi-Fi communication. Since Wi-Fi is an OTA local area network, we consider the adversary to be locally present with packet sniffing and injecting capabilities. The adversary is also able to target victim client and AP MAC addresses, spoof the AP broadcast frames on a separate channel, and create a machine-in-the-middle (MiTM) position by tricking the client to send frames to the adversary MAC address. We also assume that the adversary is able to observe all network communication in the local area without associating with any AP.

Challenges. Addressing the underlying problem of finding parsing bugs in Wi-Fi APs requires overcoming the following key technical challenges.

1) *Large input universe*: Needing to consider both the underlying protocol state and the packet structure induces a large input universe that needs to be searched for finding parsing bugs.
2) *Need for state awareness:* The SAE handshake is stateful; keys derived in one state are validated in later states. This presents a testing challenge, as each state expects specific packets. Moreover, it is possible that packets expected in a particular state trigger parsing bugs when the AP is internally in a different state.
3) *Complex packet format*: SAE handshake packets require a context-sensitive grammar to express the packet structure. Meaningfully mutating the grammar and then generating test cases from it is nontrivial.
4) *Lack of access to proprietary AP firmware*: Due to the proprietary nature of the AP firmware, any testing approach must rely on the I/O interface of the AP to derive the protocol flow and trigger deviant parsing behavior. This restriction also precludes the use of coverage-guided white-box or gray-box testing approaches. Having only access to the I/O interface also makes it challenging to infer the internal state of the AP, or whether a bug has been triggered.
5) *Need for a context-aware driver*: The driver needs to communicate with the AP for injecting the test inputs OTA. It must decode and classify the AP responses in a context-aware fashion. For example, it needs to handle anti-clogging tokens when they are presented by the AP in response to the initial driver-sent `COMMIT` packet.
6) *High cost of executing each test case*: Executing a test case OTA is expensive due to the following two reasons: (1) the driver has to wait for the AP to respond to the test input accounting for retransmissions and timeouts, and (2) the driver has to reset the AP state after each test case to ensure that the AP is in a consistent state for the next test case. The latter may require power cycling the AP.
7) *Classifying AP responses*: The absence of a reference implementation or a test oracle makes it difficult to classify whether a response from the AP constitutes an expected behavior or a parsing bug. In some benign cases, the AP's behavior may deviate from the SAE handshake specification, but it may not have security consequence. As an example, some APs send the `COMMIT` and `CONFIRM` frames together after receiving a `COMMIT` frame from the client without waiting for the client's `CONFIRM` frame.

4. Design Overview of SAECRED

SAECRED realizes the idea of combining IDS with context-sensitive grammar-based fuzzing through four main logical components: a state-aware input generator, an AP response/output classifier, the state-inference engine, and a driver. SAECRED follows an iterative testing approach and has a skeleton similar to most evolutionary fuzzing approaches. SAECRED expects two inputs: (i) the packet format expressed in a domain-specific language (DSL) we have designed by extending BNF to capture context sensitivity (*e.g.*, inter-field dependencies), dubbed ***gDSL***; (ii) a *fixed* set of initial seed inputs.

In each testing iteration, SAECRED's state-aware input generator selects some inputs from its current input population based on some scoring criteria and then mutates them to generate new inputs. It then provides these inputs to the driver, which sends these inputs OTA to the target AP. The driver waits for the AP's responses and processes them before feeding them to the output classifier and the state inference engine. The former is responsible for adjudicating the AP's responses as either *expected* or *unexpected*. The latter is responsible for inferring the state an SAE-standard-compliant AP would be in after processing the given input. We will refer to the response of the state-inference engine as the ***idealistic state***. We emphasize that this is not the actual SAE protocol state of the AP, and thus SAECRED can tolerate imprecisions in the state-inference engine.

We now discuss how SAECRED realizes this high-level workflow. An input in SAECRED is internally represented as a tuple of the form $\langle s, \sigma, g, \mathsf{score} \rangle$ in which s is the idealistic state, σ is called the state provenance, g is the gDSL representation of the SAE packet format, and score indicates the numeric score given to this input by SAECRED's scoring function signifying its utility. The state provenance σ is a sequence of symbolic and concrete SAE frames that have induced the idealistic state s. The symbolic packet in σ represents a correctly-formatted SAE packet, which may not be the expected packet in a given SAE state. The concrete packet in σ, however, is *likely* a malformed SAE packet (explained later). SAECRED partitions the inputs generated in the previous iteration based on their s values (*i.e.*, one class for each unique idealistic state). Based on the scores and some randomness, SAECRED tries to select an equal number of inputs from each class. It is also possible to bias this selection towards certain dangerous states which are likely to trigger parsing bugs. This selection of inputs from different classes ensures a fair exploration of the state space without getting stuck in the initial state.

For each of the selected inputs from the previous iteration $\langle s, \sigma, g, \mathsf{score} \rangle$, SAECRED's input generator non-deterministically decides whether to apply a mutation or induce a valid transition. If the input generator decides to mutate, it mutates the g component of the input to obtain a new grammar $g^\star$. If $g^\star$ is a well-formed grammar, it consults the SyGuS solver to generate a concrete packet that conforms to $g^\star$. Suppose the concrete packet is pkt. It then hands over $\sigma \cdot \mathsf{pkt}$ ($\cdot$ denotes the append operation) to the driver, which sends all the packets in $\sigma \cdot \mathsf{pkt}$ OTA to the AP. Based on the driver's response, the state-inference engine and the output classifier infer the new idealistic state $s^\star$ and give a verdict on the AP's response. Both of these are fed back to the input generator, which updates the score to $\mathsf{score}^\star$ and stores the input $\langle s^\star, \sigma \cdot \mathsf{pkt}, g^\star, \mathsf{score}^\star \rangle$ in the input population. In case the input generator induces a valid transition, it appends a correctly formatted symbolic SAE packet to σ and hands it over to the driver which concretizes the symbolic packet and sends it to the AP. In this case, the grammar component remains the same in the input tuple.

One natural question is why it is necessary to store both s and σ for a given input. The reason is that the idealistic state s is not the actual state of the AP. Also, σ may induce a state transition in the AP that cannot be directly mapped to any known SAE state. The detailed algorithm of SAECRED is presented in Algorithm 1 (discussed next).

5. Realization of SAECRED

This section presents how SAECRED concretely realizes the high-level design discussed in the previous section (See Figure 3). Except the driver, all SAECRED components are implemented in OCaml across $4,575$ source lines of code (SLOC). It consists of three major components, the main test generation engine (referred to as just the fuzzer from now on), a test oracle that implements both the AP response/output classifier, the state-inference engine, and a driver. SAECRED takes as input a fixed set of seed inputs and the packet format in the form of a grammar in gDSL, it and returns a set of interesting packet traces to analyze. In the rest of the section, we first discuss the main fuzzing algorithm (see Algorithm 1), followed by a discussion on SAECRED's main components: *the state-aware input generator (or, the fuzzer)*, *the test oracle*, and *the driver*.

5.1. SAECRED's Fuzzing Algorithm

SAECRED starts with a set of seed inputs. In each iteration, it samples a maximum of t inputs from the current input. It then flips a random coin (Line 12) to decide whether to mutate each input or to try inducing a valid transition (Lines 13-19). In the latter case (Lines 14-16), it picks a well-formed `COMMIT`, `CONFIRM`, or `ASSOCIATION_REQUEST` packet to append to the current input. Note that this well-formed packet may not be a desirable packet in the implementation's current state. In contrast, if it chooses to mutate the input (Lines 16-19), it first mutates the grammar component of the input to obtain a new grammar $g^\star$. It then consults the SyGuS solver (*i.e.*, `PktGenSyGuS`) to generate a concrete packet pkt that conforms to $g^\star$ (Line 18). In both cases (Lines 15 and 19), it appends the generated inputs to the next population (*i.e.*, nPop).

Each input in nPop is fed to the implementation under test using the driver (Line 21). The response packets (or, a non-response) from the AP implementation are returned (*i.e.*, π) by the `Driver`. The input σ and the obtained response π are then passed to the oracle (*i.e.*, `TestOracle`). The test oracle adjudicates (Line 22) whether the response is unexpected, in which case the input σ is added to the set of interesting traces (*i.e.*, Σ) to be manually investigated later (Lines 23-24). Finally, based on the test oracle's output, each of the new inputs in nPop is scored and stored in the current input population queue (Lines 25-26).

SAECRED repeats this process until the maximum number of iterations (*i.e.*, maxIterCount) is reached (Line 3). Every cleanupIter iterations, it also resets the input population to the seed inputs (Line 4). Also, in any iteration, if the size of the input population exceeds a maximum size (*i.e.*, maxPop), the input queue is resized to a smaller size (Line 6): 20% of maxPop.

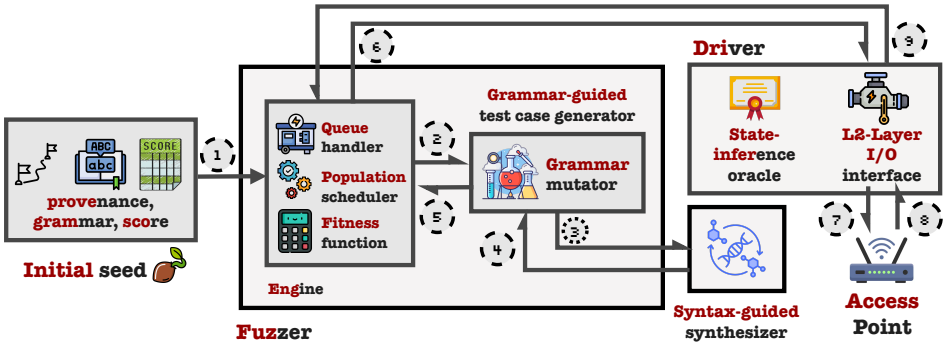


Figure 3: The complete SAECRED architecture.

Algorithm 1 The SAECRED Fuzzing Algorithm

$\Sigma \leftarrow \emptyset$ ▷ Output: Set of interesting inputs
$Q \leftarrow$ `fillStateQueues(Seeds)` ▷ Array⟨InputQueue⟩
while $i <$ maxIterCount **do**
  if i % cleanupIter $= 0$ **then**
    $Q \leftarrow$ `fillStateQueues(Seeds)`
  if `size`$(Q) \geq$ MaxPop **then**
    $Q \leftarrow$ `resize`$(Q, \frac{\text{MaxPop}}{5})$
  cPop, nPop $\leftarrow \emptyset$ ▷ Current and New Population
  for $q \in Q$ **do**
    cPop $\leftarrow$ `sample`$(q,$ `Min`$($`size`$(q), \Delta))$
  for $\langle s, \sigma, g,$ score$\rangle \in$ cPop **do**
    coin $\xleftarrow{\$} \{$`H`, `T`$\}$ ▷ Random coin flip
    if coin $=$ `H` **then** ▷ Well-formed Packet
      sym_pkt $\leftarrow$ `nondet`$($AllGoodPkts$)$
      nPop $\leftarrow$ nPop $\cup \{s, \sigma \cdot$ sym_pkt$, g,$ score$\}$
    else ▷ Mutated Packet
      $g^* \leftarrow$ `MutateGrammar`(g)
      pkt $\leftarrow$ `PktGenSyGuS`(g^*)
      nPop $\leftarrow$ nPop $\cup \{s, \sigma \cdot$ pkt$, g^*,$ score$\}$
  for $\langle s, \sigma, g,$ score$\rangle \in$ nPop **do**
    $\pi \leftarrow$ `Driver`(σ) ▷ Response packets from AP
    $\langle s^*,$ oracle_out$\rangle \leftarrow$ `TestOracle`(σ, π)
    if oracle_out is not expected **then**
      $\Sigma \leftarrow \Sigma \cup \{\sigma\}$
    score$^* \leftarrow$ `scoring`$(\sigma,$ oracle_out$)$
    $Q[s^*]$.`enqueue`$(\langle s^*, \sigma, g,$ score$^*\rangle)$
  $i \leftarrow i + 1$

Fixed Constants Used in SAECRED. SAECRED's fuzzing algorithm relies on a set of configurable constants for its operations. In our instantiation of SAECRED, we set the maximum number of iterations (denoted by maxIterCount) to 1150, the number of iterations after which the population is reset to only the seed inputs (denoted by cleanupIter) to 100, the maximum population size (denoted by maxPop) to $10,000$, the maximum number of elements picked from each queue in each iteration (denoted by Δ) to 20, and the maximum number of inputs picked overall to mutate in each iteration (denoted by t) to $3 \times \Delta$.

5.2. SAECRED's State-Aware Input Generator

We now discuss the different components behind the realization of SAECRED's state-aware input generator.

Partitioning Inputs. An input in SAECRED is a tuple $\langle s, \sigma, g,$ score$\rangle$. s denotes the idealistic state of the implementation under test according our test oracle (discussed later) when it is fed the input σ, which is a sequence of well-formed and malformed packets. We partition the inputs in the current population based on their s values (*i.e.*, one class for each unique idealistic state). We maintain a different queue for each class of inputs. In our instantiation, we have three queues for the three states of the SAE protocol: `Nothing`, `Confirmed` and `Accepted`. `Nothing` is the initial state of the SAE protocol, `Confirmed` is the state after the `COMMIT` packet is sent, and `Accepted` is the state after the `CONFIRM` packet is sent after a successful `COMMIT` message exchange. `ASSOCIATION_REQUEST` is sent in the `Accepted` state to terminate the SAE handshake and initiate the four-way handshake. Although SAECRED samples inputs from each of these queues with equal priority, it is possible to bias this selection towards certain dangerous states that are more likely to lead to interesting parsing bugs. The main goal of this partitioning is to recognize that inputs leading to different idealistic states may end up revealing different kinds of parsing bugs. In contrast, as we will demonstrate, sampling input while disregarding the idealistic state hampers both bug detection and performance.

Seed Inputs. For generating the initial seed population, we rely on the standard SAE protocol state machine. For each state s_i in the standard state machine, the seed inputs are represented as the following set: $seed(s_i) \triangleq \{\langle s_i, p, g, 0.0\rangle \mid p \in R(s_i) \wedge g \in \mathcal{G}\}$, in which $R(s_i)$ returns a set of symbolic, well-formed packets that can induce the state s_i to be reached through simple paths from the initial state, and $\mathcal{G}$ is the set of all packet format combinations in the gDSL. The seed inputs to SAECRED are $\bigcup_{s_i \in S} seed(s_i)$ where S is the set of all states in the protocol.

Sampling Inputs to Mutate. SAECRED samples from each of the state queues Δ (chosen to be 20) inputs to

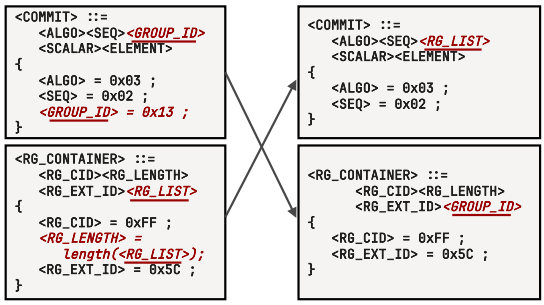


Figure 4: The crossover mutation operation. Two nonterminals are picked from two production rules and *crossed over*. Any semantic constraints on the nonterminal in the original production rule are removed.

potentially mutate (Line 10 of Algorithm 1). Out of these Δ inputs, 75% are sampled from the inputs with the highest scores, whereas the rest are randomly sampled from the entire population of that queue. In case a queue size is less than Δ, all inputs are chosen from that queue. The sampling of 75% of the highest performers from each queue is not necessarily static in the sense that SAECRED narrows down the sampling of the top percentage to prioritize the best performers (if population ≤ 100 then 50%; if population ≤ 1000 then 25%; if population ≤ 5000 then 10%; otherwise, 5%).

Mutation Operations. SAECRED mutates the grammar g of the sampled inputs to obtain a new grammar $g^\star$ (Line 17 of Algorithm 1). The mutations can be categorized into two categories: (1) structural mutations and (2) semantic mutations. The latter is introduced to violate the semantic constraints of the grammar. SAECRED non-deterministically chooses to apply one of the following mutations.

The goal of the structural mutations is to consider packet formats that are structurally different from the ones prescribed in the standard. The structural mutations are further divided into three types: *addition*, *deletion* and *two-point crossover*. In the first two types, SAECRED nondeterministically chooses a production rule and then inserts (resp., deletes) a non-terminal in the chosen production rule. The two-point crossover picks two production rules at random, two nonterminals at random, and crosses over the nonterminals in the production rules (see Figure 4). If the nonterminals are present in the semantic constraints, we delete those semantic constraints. The goal of two-point crossover is to test whether a production rule contains fields that appear out of order, leading to possible misbehavior.

The semantic mutation non-deterministically picks a nonterminal `nt` and violates the semantic constraints associated with the production rules defining `nt`. The SAE packet structure contains numerous intra-packet field dependencies such as ECC groups linked to the scalar and element or length fields of every container. The modification may apply to different types of semantic constraints, such as Boolean constraints (`length` ≤ 128) and value assignment constraints (`length` $=$ `length(rejected_groups)`). For Boolean constraints, SAECRED will take the negation of the constraint *i.e.* $\mathcal{C} \mapsto \overline{\mathcal{C}}$; for arithmetic assignments, SAECRED will nondeterministically pick either a $(+1)$ or (-1) operation and change the assignment value by the chosen arithmetic *i.e.* $\mathcal{A} = f(x) \mapsto \mathcal{A} = f(x) \pm 1$.

Identifying Ill-formed Post-Mutation Grammars. Mutating a grammar may yield an ill-formed grammar. As an example, consider a grammar with the following production rule $\langle \mathtt{S} \rangle \rightarrow \langle \mathtt{B} \rangle \mid \langle \mathtt{S} \rangle \langle \mathtt{A} \rangle$. Suppose due to the delete mutation, we end up deleting the nonterminal $\langle \mathtt{B} \rangle$ from the production rule, resulting in the production rule: $\langle \mathtt{S} \rangle \rightarrow \langle \mathtt{S} \rangle \langle \mathtt{A} \rangle$. This grammar does not generate any finite terms. To avoid situations like this, we canonicalize the grammar after each mutation (*i.e.*, ordering production rules based on their appearances), check for non-terminating circular dependencies, and discard ill-formed grammars (*e.g.*, grammars that cannot generate finite terms).

Concrete Packet Generation. For generating concrete packets from the grammar (Line 18 of Algorithm 1), we rely on a SyGuS solver [31]. The input to the concrete packet generator is a context-sensitive grammar in the gDSL, and its output is a semi-concrete packet pkt, which is concrete except for the presence of placeholders. Placeholders are special symbolic values (*e.g.*, cryptographic parameters `SCALAR` and `ELEMENT`) that are context dependent and are replaced with concrete values later. This conscious choice of leaving derived values symbolic is to ensure the termination of the SyGuS solver by precluding the non-linear reasoning that often appears in cryptographic constructs.

Due to space constraints, we explain the high-level approach using a concrete example. Suppose we want to generate a concrete, grammar-conformant packet from the following grammar in gDSL:

$$\langle \mathtt{S} \rangle \rightarrow \langle \mathtt{A} \rangle \langle \mathtt{B} \rangle \langle \mathtt{C} \rangle \ \{\ \langle \mathtt{A} \rangle < \langle \mathtt{B} \rangle || \langle \mathtt{C} \rangle\ \} \mid \langle \mathtt{A} \rangle \langle \mathtt{B} \rangle \ \{\ \langle \mathtt{A} \rangle < \langle \mathtt{B} \rangle\ \}$$
$$\langle \mathtt{A} \rangle :: \mathsf{BitVec}(16), \langle \mathtt{B} \rangle :: \mathsf{BitVec}(16), \langle \mathtt{C} \rangle :: \mathsf{BitVec}(16)$$

Above, each nonterminal symbol $\langle \mathtt{sym} \rangle$ is specified by either a *production rule* denoted by $\rightarrow$, or by a *type annotation* of the form $\langle \mathtt{sym} \rangle :: \tau$, where the type annotation means that the nonterminal symbol ranges over the given type. The expressions within curly braces denote the context-sensitive semantic constraints on the corresponding options of the production rule. We generate a corresponding SyGuS problem, formulated in the SyGuS Language [32]:

```
; "bv16" is shorthand for "(_ BitVec 16)"
(declare-datatype S ( ; algebraic datatypes
  (s_con0 (a0 bv16) (b0 bv16) (c0 bv16))
  (s_con1 (a1 bv16) (b1 bv16))))

(synth-fun f () S ; function to synthesize
; declare nonterminals
...
; grammar rules (
  (s S ((s_con0 a b c) (s_con1 a b)))
  (a bv16 ((Constant bv16)))
  (b bv16 ((Constant bv16)))
  (c bv16 ((Constant bv16)))))

(define-fun cn ((s S)) Bool ; sem. constraints
  (match s (
    ((s_con0 a b c) (bvult a (bvor b c)))
```

```
    ((s_con1 a b)   (bvult a b)))))
(constraint (cn f))
(check-synth)
```

First, we infer an *algebraic data type* (ADT) (see the blue code block) for each production rule in the grammar, which is inductively defined based on the types of the right-hand-side nonterminals. Each option in the production rule has its own constructor in the ADT, and each nonterminal within an option has its own destructor in the ADT. Second, we generate a *function to synthesize* f of type `unit` $\rightarrow$ τ (see the red code block), where τ is the inferred type of the start symbol. The grammar rules characterizing the syntactic constraints of the function to synthesize directly align with those in the input grammar. Finally, we generate an SMT-LIB [37] predicate (see the black code block) of type $\tau \rightarrow$ Bool from the given semantic constraints. We handle constraints on variables with algebraic data types by *pattern matching* on the possible constructors.

For larger grammars, we implement a divide-and-conquer algorithm that applies the above strategy to generate a SyGuS problem for each production rule with semantic constraints. Once we have the corresponding SyGuS problem(s) from g, we generate *pkt* by simply calling a SyGuS engine and serializing the results. We also replace the symbolic placeholder values with meaningful concrete values that are contextually dependent on the protocol flow.

Scoring Inputs. The scoring function (Line 25 of Algorithm 1) takes the symbolic output of the `TestOracle` and scores the input. The four symbolic outputs are `timeout`, `expected_output`, `crash`, and `unexpected_output` with the weighted scores of 0.1, 0.3, 0.5 and 0.7, respectively. The weights reflect the preference of the classes of bugs that SAECRED targets.

Packets that lead to timeouts are scored the least because such inputs can tremendously slow down the fuzzing process and may not necessarily lead to interesting bugs unreachable from other inputs. Although expected outputs are not very interesting, they are preferred over timeouts because they may lead to interesting bugs in the future. Crashes and unexpected outputs are scored the highest since crashes lead to complete DoS of APs, and unexpected outputs imply parsing bugs with possible security implications.

5.3. SAECRED's Test Oracle

SAECRED's test oracle is responsible for classifying whether the AP's responses to a given input are desirable. It is also responsible for inferring the idealistic state of the AP under test after processing a given input. It mainly consists of two components: a packet parser and a protocol state machine. Note that we follow the standard to manually construct the test oracle. SAECRED's effectiveness does not necessarily rely on the correctness of the test oracle. However, a correct test oracle can improve the effectiveness of SAECRED.

5.4. SAECRED's Driver

SAECRED's driver consists of $2,909$ lines of Python code. It communicates with the main testing algorithm. It carries out two tasks: (1) replacing placeholders with contextually meaningful values and (2) sending and receiving packets over the link. The driver maintains the cryptographic context and SAE state, allowing SAECRED to transparently and statefully interact with the APs under test.

After running each test case, the AP state must be reset. Prior work [2] sent `DEAUTHENTICATION` frames to reset the AP state. We observe that COTS APs often do not respect this behavior. An alternative reset approach is to complete the SAE handshake and let the four-way handshake time out. This strategy, however, requires waiting for the reset timer to expire (*i.e.*, 7 seconds). In some COTS APs, this approach also does not work. When an AP has an administration portal, we use the Selenium WebDriver and instrument Selenium to connect to the AP web interfaces and programmatically restart the APs without removing the WPA3-Personal-only configuration, SSID, and passphrase. For some APs, we use smart switches to programmatically power cycle the APs when none of the above approaches worked.

The AP's internal SAE configuration is opaque to the driver. Whether APs support `H2E`, multiple `ECC` groups, or password identifiers is unknown to the driver. Thus, a generalizable driver can only make a best effort in attempting to report test cases as deviating. For example, consider an AP implementation supporting only one ECC group. If the driver were to send a packet that contains a stronger, AP-unsupported ECC group in the rejected groups container, the AP will not respond with an error, as it does not support the rejected group. The driver will report this case as a deviation, as it is oblivious to the internal AP configuration. These edge cases require manual investigation into the nature of the test cases and require a separate root-cause analysis.

6. Evaluation

This section presents the evaluation details of SAECRED, showcasing its effectiveness in identifying deviant protocol behaviors and comparing it against baseline approaches.

Test Subjects. We evaluate 6 COTS APs that support the SAE handshake and the open-source `hostapd` implementation. Details of these APs, including their firmware versions, are provided in Table 6. Among these, 3 APs support the new hash-to-element algorithm, which incorporates optional containers for enhanced security guarantees.

Preparing APs for Testing with SAECRED. To evaluate the SAE handshake, each AP is manually configured to operate in WPA3-Personal mode exclusively. By default, APs are preconfigured for WPA2-Personal, necessitating a switch to WPA3-Personal to focus on the updated SAE handshake. This ensures that (1) the fuzzer's scoring function is not skewed by WPA2-related test cases, and (2) transition-mode vulnerabilities, which are well-documented to have weaker security guarantees [2], are avoided. The primary objective

is to uncover new bugs within the SAE handshake rather than in transition-mode operations.

6.1. Research Questions

The evaluation of SAECRED is guided by the following research questions:

- **RQ1.** How well does SAECRED find parsing bugs?
- **RQ2.** How effective are SAECRED's mutation operations in generating meaningful test cases?
- **RQ3.** What is the contribution of each individual component of SAECRED to its overall efficacy?
- **RQ4.** How does SAECRED compare to baseline approaches in terms of code coverage?

Setup for RQ1-RQ3. We adhere to established fuzzing best practices [38] across all test subjects. SAECRED's efficacy is evaluated based on the number of deviations identified.
Answering RQ2. We analyze the discovered bugs and trace them back to the responsible mutation operations. This provides insights into the effectiveness of semantic constraint modifications and structural modifications in generating meaningful test cases.
Answering RQ3. To assess the contributions of SAECRED's individual components, we perform a systematic decomposition. This involves selectively disabling key functionalities, such as grammar-based mutation versus byte-level mutation, multi-queue versus single-queue setups, and state-inference-guided feedback versus temporal scoring. By comparing the results of these configurations, we quantify the importance of each component in improving fuzzing outcomes.
Answering RQ4. We utilize coverage data obtained from all baseline and ablation campaigns conducted on `hostapd`. This allows us to measure SAECRED's impact on achieving higher SLOC coverage compared to alternative approaches.

6.2. Baselines

To establish a fair comparison, we evaluate SAECRED against 3 baseline configurations. Each baseline experiment is conducted over 5 trials, with each trial lasting 24 hours. Coverage is measured using `lcov` for the open-source `hostapd` implementation.
AFL++ with grammar-based fuzzing. We implement custom handlers for `hostapd`'s `read()` and `recv()` functions and run AFL++ using a context-free grammar (CFG) approximation of SAECRED's input grammar. Semantic constraints are removed to align with AFL++'s CFG-based mutation capabilities. The generated inputs are passed directly to `hostapd`, bypassing socket-level system calls for improved performance. Crashes and memory corruption bugs are recorded, and coverage is compared with SAECRED.
SAECRED with AFL's Byte-Level Mutations. SAECRED integrates AFL's `havoc` mutator to perform byte-level mutations on the input data. These mutations are applied to inputs generated by SAECRED's context-aware grammar. The resulting test cases are then passed to `hostapd` in a black-box testing setup. We analyze the deviant traces and coverage metrics to assess the impact of combining AFL's byte-level mutation strategy with SAECRED's grammar-based approach, comparing the results against the original SAECRED implementation.
SAECRED with AFL++ grammar-mutation operations. We configure SAECRED to use AFL++'s grammar mutator on a CFG representation of the input grammar. Semantic constraints are removed to ensure a fair comparison. Deviant traces and coverage metrics are analyzed to evaluate the significance of SAECRED's context-sensitive approach.

6.3. Ablation Study

To further evaluate SAECRED's design choices, we conduct two ablation experiments on `hostapd`, each consisting of 5 24-hour trials.

1) **Single population queue.** We configure SAECRED to use a single population queue instead of state-specific queues. Candidates are sampled from this unified queue, and the resulting deviant traces and coverage metrics are analyzed.
2) **Temporal scoring for feedback.** We replace SAECRED's state-inference-based scoring with a temporal scoring function, which prioritizes test cases based on execution time. This approach, inspired by prior works [21], is evaluated to determine the utility of state-transition-based feedback. Deviant traces and coverage metrics are collected for analysis.

These experiments will answer RQ3 by providing a comprehensive understanding of SAECRED's capabilities and its advantages over existing methods.

7. Findings

We will now answer the RQs in light of the findings from the evaluation. We will discuss the parsing bugs found, the efficacy of the holistic SAECRED approach (RQ1), the need for each of SAECRED's functionalities (RQ2-RQ3), the coverage achieved (RQ4), and the runtime performance. Overall, SAECRED uncovered bugs belonging to 4 distinct classes. The classes are as follows, namely: (1) *Downgrade bugs*, (2) *State-poisoning bugs*, (3) *Unexpected successful association bugs*, and (4) *General parsing bugs*.

7.1. Semantic constraint violation (RQ2)

SAECRED's mutations lead to semantic constraint violations in the APs under test. The most notable and high impact violation was the *downgrade bug*, where a malformed rejected groups container leads to a successful downgrade on the ECC group, despite the client and AP supporting a strong ECC group. The bug is a result of SAECRED's arithmetic modification operator described in Section 5 that reduces the length of the rejected groups container by 1, bypassing a critical check. The APs under test are expected to reject the commit frame and terminate the handshake, but SAECRED found that all APs supporting `H2E` and at least two ECC groups responded with `COMMIT_SUCCESS` frames.

One would assume that the cryptographic primitives would be able to detect this violation, but Section 9 shows that the parsing bug leads to the same key generation steps on both the client and AP sides, resulting in an undetectable and complete downgrade of the authentication handshake with both parties using weaker ECC groups. This is a violation of the standard and breaks the downgrade protection guarantees. We will discuss the security implications and concrete attack vector in Section 9.

To answer RQ2, SAECRED*'s semantic constraint mutation triggers a security-critical parsing bug resulting in a guarantee-breaking downgrade attack. Context-sensitive grammars and modifications to the constraints are necessary to identify this bug.*

7.2. Structural violation (RQ2)

SAECRED's structural mutations uncover a number of parsing bugs that either have direct security implications or lead to unintended system state transitions. The most notable structural violation was the *state-poisoning* bug that results in a long-term client DoS attack. The structural violations come in three flavors of bugs, namely (1) the state-poisoning bug resulting in password-identifier-based DoS attacks, (2) unexpected successful association bugs, and (3) general parsing bugs.

While these bugs underscore the significance of SAECRED's mutation operations (RQ2), they also show the significance of SAECRED's state awareness (RQ1). The test packet leads to two unexpected behaviors: (1) A commit frame with a password identifier container is accepted by some APs despite not being configured for password identifiers, and (2) any valid commit frame sent after the test packet, but lacking the password identifier, times out. Both of these behaviors are captured by the state-inference engine as it reports that both of these behaviors are unexpected. This is another high-impact bug: the password identifier DoS discovered by SAECRED represents the state-of-the-art WPA3-SAE DoS attack, in which an attacker can trigger a long-term denial of service on the client by injecting a single frame without expending significant resources. We will describe the concrete attack vector and the underlying root cause in Section 9.

The impact of the unexpected successful association and general parsing bugs is unknown to us, and figuring this out is an ongoing effort. SAECRED observed these unexpected behaviors, however, at first glance, the security implications are not clear. We will discuss these bugs in detail below and delve into the high-impact vulnerabilities in Section 9.

Unexpected successful association. This class of parsing bugs results in the APs under test accepting malformed commit frames and entering the `Accepted` state. This is a violation of the standard, as the APs under test are expected to reject the commit frame and terminate the handshake. However, SAECRED found that certain APs under test transitioned to the `Accepted` state despite receiving malformed commit frames. This behavior violates the standard and indicates a parsing bug that could potentially lead to undefined protocol behavior or security vulnerabilities, such as buffer overflow vulnerabilities. Further investigation is required to determine the full impact of this issue.

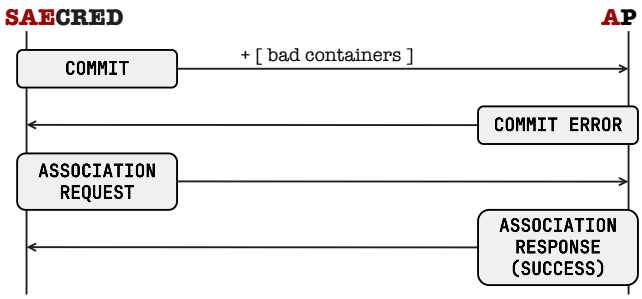


Figure 5: Protocol flow violation leading to an invalid state transition to the accepted state.

This behavior is evident in Figure 5, where SAECRED is able to associate with the AP without ever entering the confirm phase of the handshake. This flow shows that given a commit frame with bad containers, it is possible to jump directly into the `Accepted` state without sending a confirm frame indicating that both parties generated the same `PMK`. Typically, the four-way handshake begins, but due to the invalid flow, the AP does not initiate the four-way handshake. We are unsure about the security implications of this undefined protocol flow, as SAECRED is not configured to fuzz the four-way handshake. However, we cannot rule out the possibility of significantly worse behavior by some APs. At the very least, parsing invalid containers opens the possibility of buffer-overflow bugs, but it is difficult to confirm in a black-box setting with no binary source to instrument.

General bugs. The final class of parsing bugs SAECRED found are general bugs that may or may not have security implications. SAECRED reported test cases that were malformed due to packet structure violations, which nevertheless led to successful state transitions. The state-inference engine reported that these packets are invalid and should not trigger state transitions, therefore making their successful parsing unexpected. Empirical evaluation of these bugs showed no direct security impact; however, we cannot rule out the absence of bugs or buffer-overflow vulnerabilities that result as a consequence of parsing malformed packets.

This class of parsing bugs is distinct from the unexpected association bugs, as in the former, the APs under test transition to the `Accepted` state, while in the latter, the APs under test simply accept malformed SAE frames. In the context of SAE, the unexpected association bugs are more severe as they signal successful association with the AP, while the general parsing bugs may not lead to any significant impact. However, both classes of bugs highlight the need for robust parsing and validation mechanisms in the SAE protocol to prevent unexpected behavior and potential security vulnerabilities.

To answer RQ2, SAECRED*'s structural mutation operations trigger a number of stateful parsing bugs that lead to*

unexpected system behavior. All of these bugs are a result of the structural mutations on the input grammar and the state-inference engine's ability to identify the unexpected behavior. Without the composition of the context-sensitive grammar and the state-inference engine, these bugs would not have been identified.

Capability	Downgrade	DoS	Associate	General
Intercept	✓	✓	✓	✓
Modify	✓	✓	✓	✓
Impersonate	✓	✓	✓	✓
Drop	✗	✓	✗	N/A
Inject Malformed	✓	✗	✓	✓
Key Compromise	✗	✗	✗	✗

TABLE 2: Adversary capabilities required for each attack.

8. SAECRED's multifaceted approach

We now ascertain the efficacy of the SAECRED architecture down to the significance of each individual component. To answer RQ3, we will show that each component adds to the required functionality needed to find deviant behavior in the SAE handshake, and that the combination of all components is necessary for the high-coverage and high-quality test cases that SAECRED produces. To answer RQ4, we will show that despite running in black-box mode, SAECRED is able to cover more lines and functions than the baselines and ablations. The detailed answers with empirical evidence to RQ2-RQ4 help answer RQ1 regarding the holistic effectiveness of SAECRED in finding parsing bugs.

8.1. Baseline and ablation comparison (RQ3)

Fuzzer	Downgrade	DoS	Associate	General
SAECRED	16	8652	119	1114
SAECRED SQ	30	6053	–	438
SAECRED SQ + T	–	6771	–	530
SAECRED AFL	–	–	N/A	–
SAECRED AFL++	–	–	N/A	–

TABLE 3: Identified bug instances of each category in ablation study. SQ: Single-queue, T: Temporal scoring function.

Table 3 shows the number of instances of each bug found across the ablation experiments. We analyze the results to understand the significance of each SAECRED component.

SAECRED uncovers all bug types with higher instances. SAECRED consistently identifies numerous instances of all discussed bug types. The only exception is the downgrade bug, where it finds fewer instances due to its multi-queue design. SAECRED aims to explore the state space comprehensively, ensuring fair traversal and bug discovery across all protocol states. This approach prioritizes parsing bugs in the `Nothing` state equally with those in subsequent states. Consequently, SAECRED is uniquely equipped to uncover the association bug, which requires fuzzing later states in the protocol flow.

Single-queue exhibits an initial state bias. The single-queue ablation fuzzer finds more instances of downgrade bugs but identifies fewer instances of other bugs, with no association bug instances. This behavior indicates that a single-queue SAECRED prioritizes bug hunting in the initial state, as both the DoS and downgrade bugs are reachable in the `Nothing` state. This is evident from the fuzzer's inability to find any instances of the association bug.

Temporal scoring is insufficient. Out of various SAECRED configurations with the same mutations as vanilla SAECRED, the temporal scoring configuration performs the worst. It is unable to reason about the possible state transitions as feedback, and as a result, it converges to prioritizing test cases that cause timeouts, indicating test case wastage. To better guide the fuzzer towards possible parsing bugs, the fuzzer must rely on the state-inference engine's feedback.

Grammar-aware mutations surpass byte-level mutations. SAECRED AFL was equipped with a valid WPA3-SAE grammar, which the SyGuS solver and driver translated into valid SAE byte sequences. These sequences were then mutated using AFL's bitwise havoc mutator. However, the havoc mutator operates in a non-deterministic manner, disregarding the cryptographic and inter-field dependencies inherent to the protocol. As a result, the generated test cases were malformed, leading to frequent timeouts or rejections by the APs. This lack of adherence to protocol constraints results in zero instances of bugs, highlighting the importance of context-sensitive grammars in effective fuzzing.

Context-sensitive grammar surpasses context-free grammar. SAECRED AFL++ utilized the context-free variant of the WPA3-SAE grammar; it applied mutations similar to AFL++'s grammar mutator. However, the absence of context meant the fuzzer could not preserve the cryptographic and inter-field dependencies. This results in malformed packets that are either dropped or rejected by the APs, resulting in minimal coverage and no bug discovery. Conversely, the context-sensitive grammar ensured that generated packets conformed to protocol constraints, allowing the fuzzer to explore deeper protocol states for parsing bugs.

Answering RQ3, each individual SAECRED *feature plays a critical role in its bug-finding capability. The removal of any feature, whether related to mutation strategies or system architecture, significantly diminishes* SAECRED*'s ability to find bugs, underscoring the necessity of its holistic design.*

8.2. Coverage analysis (RQ4)

For coverage analysis, we delve into the lines and functions uncovered by SAECRED and compare them with the baseline and ablation experiments. This analysis provides insights into the effectiveness of SAECRED in exploring the SAE parser handler code.

We begin by examining Table 4, which presents the coverage results for two critical SAE parser handler source files: `sae.c` and `ieee802_11.c`. These files are pivotal as they handle SAE packet parsing. The table distinguishes between overall coverage and restricted coverage within these files. This distinction is necessary because the `hostapd` harness

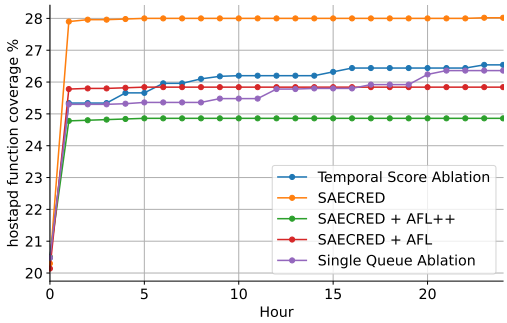


Figure 6: Average coverage for SAECRED, baselines and ablation studies.

lacks access to the complete network stack, particularly code referencing socket calls. To ensure a fair comparison with the custom test harness, we focus on the specific portions of the code responsible for SAE packet parsing.

Gray-box AFL++ achieves the lowest coverage. AFL++ with the grammar mutator running in gray-box mode exhibits the least coverage of the SAE parser handler code. Despite being initialized with a seed representative of valid SAE interactions, the context-free nature of the grammar mutator results in the loss of SAE-specific context. This limitation prevents the fuzzer from making significant progress compared to the black-box SAECRED configuration.

SAECRED with AFL++ mutations achieves limited black-box coverage. When SAECRED is configured with a context-free grammar, it performs better than AFL++ due to the inclusion of cryptographic placeholders. However, it still falls short compared to configurations utilizing context-sensitive grammars. AFL, equipped with context-sensitive grammars, mutates the generated byte sequences more effectively, uncovering additional code. This highlights the critical role of capturing inter-field dependencies and cryptographic context in achieving higher coverage.

SAECRED achieves the highest coverage. SAECRED, leveraging its multi-state queue architecture and oracle-guided feedback functions, achieves the most extensive function coverage across the SAE parser handler files. The combination of intelligent population selection, fine-tuned mutation operations, state awareness, and feedback mechanisms enables SAECRED to explore deeper program states and uncover more code.

SAECRED demonstrates the fastest coverage growth. As illustrated in Figure 6, SAECRED exhibits rapid growth in function coverage compared to other configurations. It quickly reaches peak coverage, while other configurations progress at a slower pace. Notably, even after running for 24 hours, all baselines and ablations fail to converge to the peak coverage SAECRED achieves in just 1 hour. This underscores the effectiveness of SAECRED in exploring the SAE parser handler code comprehensively.

Addressing RQ4, SAECRED *consistently achieves the highest SLOC coverage at the fastest rate compared to both baselines and ablation campaigns. This demonstrates that even in black-box mode, context-sensitive input generation combined with state-inference-guided feedback effectively reaches SLOC that would otherwise remain unexplored.*

	Line(%) `sae.c` `ieee802_11.c`	**Function(%)** `sae.c` `ieee802_11.c`
SAECRED	55.0 41.5	75.9 63.3
AFL++ gray-box	10.7 12.1	20.4 15.2
SAECRED AFL	53.6 28.7	75.9 46.8
SAECRED SQ	54.5 40.3	75.9 63.3
SAECRED SQ + T	54.7 37.3	75.9 60.8
SAECRED AFL++	42.0 23.4	68.5 39.2

TABLE 4: Maximum line and function coverage for all baseline, ablation, and gray-box mode experiments. The top value in each fuzzer row corresponds to `sae.c`, and the bottom one corresponds to `ieee802_11.c`. SQ: Single queue, T: Temporal scoring function.

8.3. Performance

We now discuss SAECRED's network and test case generation performance in terms of temporal measures. Table 5 presents the average time SAECRED takes across all APs under test to execute a test case, complete a fuzzing iteration, populate each queue fully, and handle malformed grammar-to-packet responses. On average, executing a trace takes $\geq$ 12.1s. While the latency is high, the coverage achieved by SAECRED justifies this cost. Although testing a trace requires more time, the results demonstrate the high *quality* of the generated test cases.

Additionally, SAECRED completes an iteration in $\geq$ 742.7s on average and requires $\geq$ 7 iterations to populate all queues. Initially, SAECRED exhibits a bias toward the `Nothing` state, but this bias diminishes as it continues to run, enabling effective exploration of the state space. This behavior highlights the capability of SAECRED's architecture to address initial state bias effectively.

Lastly, SAECRED fails $\leq$ 14.7 times on average and spends $\leq$ 3.3 seconds waiting for malformed grammar-to-packet responses. Over a 24-hour run, this cost is negligible given the high coverage and the discovery of issues across *all* `H2E`-supported COTS APs.

9. Attacks

This section presents two of the representative attacks (*i.e.*, downgrade and DoS) SAECRED uncovered. The adversarial assumptions needed for these attacks are discussed in Section 3 and shown in Table 2.

	hostapd	ASUS-TUF	ASUS RT	TP-Link AX3000	TP-Link AX21	eero	Verizon
Avg. single test execution time	14.6	16.3	16.1	13.8	28.24	12.6	12.1
Avg. time to run a single fuzzing iteration	953.3	1438.6	1419.9	1230.2	2441.6	741.7	1096.0
Number of iterations for filling all queues	8.8	8.2	8.4	7.0	7.4	6.7	9.2
Number of total times SyGuS failed	11.4	13.4	14.2	13.6	5.0	14.7	14.6

TABLE 5: Average performance breakdown over 5 24-hour trials.

AP	Chipset	Firmware	H2E	Downgrade	DoS	Associate	General bugs
ASUS-TUF AX6000	MediaTek MT7986AV	3.0.0.4.388_33405	✓	✓	✗	✓	✓
ASUS RT-AX1800S	MediaTek MT7621AT	3.0.0.4.386_69041	✓	✓	✗	✓	✓
TP-Link Archer AX3000	Intel GRX350A3	1.1.2	✓	✗	✓	✓	✓
TP-Link Archer AX21	Broadcom BCM6755	1.1.0	✗	NA	NA	✗	✗
Amazon eero 6	Qualcomm IPQ8174	7.6.2-130	✗	NA	NA	✗	✗
Verizon Fios CR1000A	Qualcomm IPQ8072A	3.2.0.14	✗	NA	NA	✗	✗
hostapd	NA	2.10	✓	✓	✓	✓	✓

TABLE 6: APs tested and their susceptibility to different attacks. NA refers to an AP not supporting the H2E algorithm.

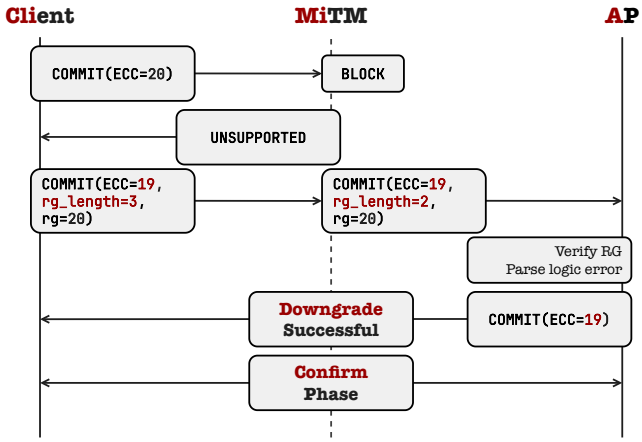


Figure 7: The H2E rejected groups downgrade attack.

```
1267  static int check_sae_rejected_groups(struct
      hostapd_data *hapd,
1268      struct sae_data *sae)
1269  {
1270      const struct wpabuf *groups;
1271      size_t i, count;
1272      const u8 *pos;
1273      if (!sae->tmp)
1274          return 0;
1275      groups = sae->tmp->peer_rejected_groups;
1276      if (!groups)
1277          return 0;
1278      pos = wpabuf_head(groups);
1279      count = wpabuf_len(groups) / 2; //PARSE ERROR
1280      for (i = 0; i < count; i++) { //PARSE ERROR
1281          int enabled;
1282          u16 group;
1283          group = WPA_GET_LE16(pos);
1284          pos += 2;
1285          enabled = sae_is_group_enabled(hapd,
      group);
1286          wpa_printf(MSG_DEBUG, "SAE: Rejected
      group %u is %s",
1287                  group, enabled ? "enabled" : "
      disabled");
1288          if (enabled)
1289              return 1;
1290      }
1291      return 0;
1292  }
```

Listing 1: hostapd SAE rejected groups validation source code in ieee802_11.c. Purple text shows the parsing bug.

9.1. Downgrade attack

In this attack, the adversary's goal is to downgrade the ECC group used in the SAE handshake. The downgrade attack is realizable due to implementations *improperly* validating the rejected groups' container length in the SAE commit frame. SAECRED generated a test case in which the commit frame contained a single rejected groups container with the rejected group 20 and the length set to 1. Two COTS APs and hostapd successfully parsed this *malformed* rejected groups container and continued with the handshake. This parsing bug allows an adversary to launch a machine-in-the-middle (MiTM) attack and force the AP to downgrade to a weaker security group without the client's knowledge.

Downgrade defenses. The standard defines the following defenses to detect the downgrade and make the attack infeasible: (1) checking the ECC groups in the rejected groups container; (2) using the values in that container in the PMK KDF. This makes trivial packet manipulation attacks infeasible as an MiTM adversary cannot simply remove the container and hope for the handshake to succeed. While both parties will successfully move on to the confirm phase, the confirm phase fails due to the different keyseeds used in the respective KDFs. To launch the downgrade attack, the adversary must ensure that both parties use the same keyseed for their respective KDFs.

Downgrade attack flow. Figure 7 shows the downgrade attack. The adversary intercepts the commit frame from the client, drops the frame, and impersonates the AP by responding with an UNSUPPORTED_CYCLIC_GROUP error. The client downgrades to a weaker ECC group (*e.g.*, group 19), uses the stronger ECC group (*e.g.*, group 20) as the

salt for the KDF, and appends the stronger ECC group to the rejected groups container. The adversary intercepts this commit frame and reduces the length of the container by 1. The AP accepts this frame, uses the strong ECC group that *it supports* as the salt for the KDF and proceeds with the handshake. Both parties now use the same salt for the KDF, and therefore both sides derive the same PMK.

Root-cause analysis. We investigate this abnormality by analyzing the hostapd source code. Listing 1 shows the rejected groups validation code, in which line 1279 extracts the length of the rejected groups container and integer divides by 2. This is problematic as valid rejected group lengths are always even, but in the event that the length is odd, the loop on line 1280 terminates without iterating. This means that the code on line 1284 is never reached and the rejected groups container is never validated.

Interestingly, we note that this attack is possible not only because of preemptive loop termination, but also due to the HMAC function used in the KDF. Suppose the client appends the rejected group 20 to the commit frame. The hexadecimal representation of this is 14 00. When the adversary modifies the length to 1, the AP processes the rejected groups container value as 14, missing the 00 byte that the client is using as the salt for the KDF. We note that despite this difference, the handshake still succeeds as the HMAC function used in the KDF *pads a zero byte in the event that the input is less than 2 bytes.*

9.2. DoS attack

The adversary's goal with this attack is to poison the AP's internal state by sending a commit frame with a password identifier container with arbitrary password identifier values (PassID), resulting in a long-term DoS attack on the client. The root cause of the DoS is that subsequent commit frames without the same PassID container are dropped by the APs. This is an acceptable behavior because honest clients using PassID will append the same container to subsequent commit frames. However, in the context of an adversary, this behavior is problematic, as the adversary can simply append a PassID container with arbitrary values and cause the AP to accept the container. Since the client is unaware of the PassID container, the client will not append the container to subsequent commit frames, causing a DoS.

SAECRED discovered this bug due to two reasons. First, SAECRED appended PassID containers with arbitrary values and expected the AP to reject the commit frame. The vulnerable APs, however, accepted the commit frame and continued with the handshake. These frames were marked by SAECRED as *unexpected*. Secondly, in subsequent commit frames, SAECRED either did not append any PassID container or appended a different PassID container. These frames were dropped by the AP causing SAECRED's state-inference engine to report unexpected behavior.

DoS attack flow. Figure 8 shows the DoS attack. The adversary intercepts the SAE commit frame from the client, appends a PassID container with arbitrary values, and forwards the modified frame to the AP. The AP accepts this frame and continues with the handshake. Additionally, the AP updates its internal state with the PassID values. It then responds to the client by sending its own commit frame with the PassID chosen by the adversary. The adversary must intercept and block this frame, as the client must not be made aware of the PassID values. Once the adversary blocks the AP-sent commit frame, the attack is complete and the adversary can drop off the network.

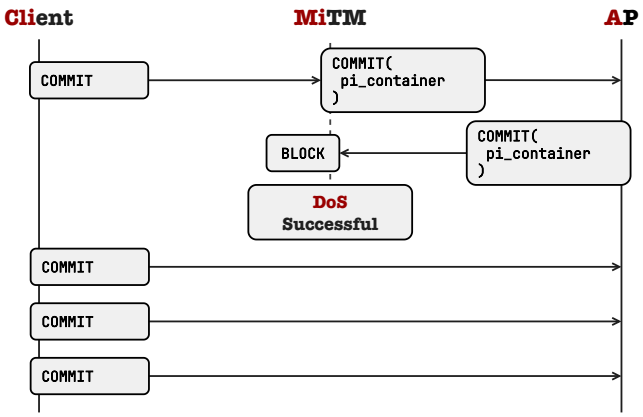


Figure 8: Long-term DoS attack through a single commit frame. Clients are indefinitely DoSed.

Root-cause analysis. The DoS attack is possible due to (1) acceptance of arbitrary PassIDs even in the event that APs are not configured for PassIDs and (2) APs' management of internal timers for new clients. Whenever APs receive commit frames, they initialize a timer for the authentication session. If the timer expires, the AP clears the client state. This implies that whenever an authenticating client sends a commit frame, the timer resets. An adversary exploits this behavior by sending a commit frame with arbitrary PassIDs and then dropping the AP-sent commit frame. Since clients are (1) unaware of this modification to their commit frame and (2) assume the commit frame was never received by the AP, they begin sending new commit frames in an attempt to authenticate. The APs expect the PassID appended by the adversary to be present in these new commit frames, and since they are not, they drop the commit frames and reset the authentication session timer. As a result, if the client attempts to repeatedly authenticate, the AP will never clear the client state as the timer resets and the AP will continue to maintain the poisoned PassID variable for the authentication session.

To answer RQ1, SAECRED *identified two high-impact vulnerabilities due to its multi-state queue architecture, oracle-guided feedback, and state-inference engine. These high-impact vulnerabilities were not identified for several years, which shows* SAECRED*'s ability to uncover previously unknown vulnerabilities.*

9.3. Responsible disclosure

All bugs and their exploitations were disclosed to all relevant stakeholders. Several vendors have acknowledged

the vulnerabilities, launched patches, and sent security advisories, and others are still working on remediation at the time of this writing. `hostapd` has received a patch for the downgrade attack on the stable branch and the `PassID` DoS attack on the development branch [39], [40]. MediaTek issued CVE-2025-20664 [41]. While the attacks are found on `hostapd`, patches were issued both to `hostapd` and `wpa_supplicant`, indicating that clients were also prone to these attacks.

10. Related Work

We now categorize existing protocol implementation testing efforts and show how they differ from SAECRED.

Protocol-specific and -agnostic Testing. A majority of the testing approaches can be categorized either as general protocol implementation testing approaches [16], [11], [15], [42], [43], [44], [19], [45], [12], [46], [47], [48], [13] or as approaches that are tailor-made for a specific protocol [10], [49], [14], [22], [18], [23], [50], [27], [51]. SAECRED is tailor-made for the WPA3 SAE handshake and includes aspects that are not necessarily generalizable.

Additional Inputs to Testing Approaches. Various testing approaches require access to the implementation under test (***IUT***). Additionally, there are approaches that require the abstract protocol state machine [14], [50], [45], [15], [17], [46], a diverse set of implementations [17], [20], a set of desired properties [24], [25], [16], and a packet format specification [13], [51], [21], [50], [23], [14], [10] as additional inputs. SAECRED requires packet formats, the protocol state machine, and a parser for the SAE handshake.

Levels of Access to the IUT. One of the clearest distinctions between the various testing approaches is the level of access they have to the IUT. The primary categories are: (1) black-box access [18], [15], [45], [23], [20], [48], [47], [50], [21], [17], [51], (2) gray-box access [10], [11], [49], [14], [22], [12], [13], [46], [44], [19], [29], and (3) white-box access [43]. Black-box access is the most restricted having only access to the input and output interface. The other extreme is the white-box access, which has full access to the source code of the IUT. Finally, gray-box access is a middle ground between the two extremes, where the IUT can be instrumented to observe coverage information. SAECRED adopts the black-box approach, which allows it to be applicable to any commercial AP.

State Awareness. Stateful protocol testing is challenging as fuzzers need to maintain state with the IUT. This is one of the main distinguishing aspect of testing a stateful system [10], [11], [49], [14], [22], [12], [46], [44], [15], [45], [23], [48], [47], [50], [17], [51] compared to a stateless one [13], [19], [29], [18], [20], [21]. Many of the existing approaches do maintain some form of state awareness; however, their state exploration is guided by additional information such as the coverage information or the precise state of the IUT. SAECRED explicitly maintains states but does not rely on any additional information for its state exploration.

IUT Instrumentation. Many existing gray-box fuzzers rely on instrumentation of the IUT to obtain coverage information, crash status, or precise IUT state inference [10], [11], [49], [14], [12], [13], [46], [44], [19], [29]. These rich forms of feedback can effectively guide testing to find interesting bugs. SAECRED does not rely on any information that requires instrumenting the IUT.

State Inference. Many existing testing approaches rely on inferring the underlying protocol states of the implementation. The primary approaches for state inference include L*-style state machine inference [17], source code analysis [52], [44], implementation instrumentation for precise state tracking [12], [11], [44], status code analysis [15], [47], heuristic extraction from memory snapshots [46], [11], and oracle-based state inference [10], [11], [22], [12], [23], [51], [23]. SAECRED relies on an oracle to infer the underlying protocol state but is aware that the inferred state may not be the precise implementation state.

Resetting States. Due to statefulness, it is paramount to reset the protocol state after running each test case. The predominant reset approaches are (i) restoring from a snapshot [46], [11], (ii) soft/hard device resetting (*e.g.*, power cycling), and (iii) applying a homing sequence [17]. SAECRED relies on a combination of (ii) and (iii). Approaches that rely on memory snapshots to restore/reset protocol states enjoy a significant speed advantage over other approaches. As a result, SAECRED is slower than snapshot-based fuzzers and hence focuses on only executing effective test cases with a higher likelihood of triggering bugs.

Packet Format Awareness. As discussed before, payload formats add an extra dimension to the challenges of testing protocol implementations. However, not all protocol messages are equally complex. Some message formats are simple enough to be captured with a context-free grammar [28], [13], [21], [19], [29], whereas others require the power of context sensitivity. In addition, some message fields may have derived relationships that are sensitive to the protocol flows (*e.g.*, cryptographic constructs). Many existing fuzzers can only handle packets that are context free whereas SAECRED can handle packets that are context sensitive with protocol flow dependencies.

Mutation Operations. Many existing fuzzers still rely on bitwise mutation to generate new test cases [10], [11], [49], [12], [46], [44], [18], [15], [45], [20], [48], [50], [47], [51]. Those that explicitly consider the packet format often use a grammar to generate well-formed inputs, and then apply mutations to the generated inputs. Such approaches often do not end up exercising critical parts of the protocol implementation because the mutations are not format aware. In contrast, approaches such as ATFuzz [21] directly mutate the format (*i.e.*, grammar production rule) to generate new test cases. SAECRED follows the latter approach but also maintains the state-awareness of the explicit protocol.

Crash and Logical Bugs. Finally, many existing fuzzers mainly target crash bugs in implementations [13], [46], [44], [19], [29], [18], [15], [47], [50]. These approaches often cannot detect logical bugs, which are challenging to discover. Existing approaches either use differential testing of multiple implementations [17], [20], [14] or rely on an oracle to detect logical bugs [10], [11], [22], [12], [23].

SAECRED relies on an oracle to detect logical bugs.

11. Discussion

Limitations. While SAECRED pushes the boundaries of Wi-Fi fuzzing with an approach tailored to the SAE handshake, it remains far from an *ideal* general-purpose Wi-Fi fuzzing tool (*e.g.*, testing the 4-way handshake).

To SAECRED, fuzzing is a means of generating inputs, approximating internal state, and using symbolic values to evaluate the quality of test cases. This approach assumes the existence of a driver capable of mapping SAECRED's symbolic values to concrete, context-aware cryptographic values, while also maintaining protocol state across the entire test sequence and making subtle flow decisions — often in ways that are opaque to SAECRED. This dependency makes extending SAECRED to other Wi-Fi sub-protocols difficult.

SAECRED relies on the driver to maintain a state with the AP and to make *reasonable* approximations about the expected protocol state. This requires manual interpretation of often ambiguous specifications, leaving behavior open to the implementers' interpretation. Therefore, developing a reliable oracle is challenging. Even if one is implemented, some APs may behave differently due to vendor-specific quirks.

Finally, traces flagged for further analysis still require manual inspection to determine the root causes. While the grammar aids in debugging, understanding the origin and exploitability of a parsing bug remains a manual task.

Lack of AP specification. A major challenge in fuzzing COTS APs is the lack of transparency. Vendors may advertise WPA3 support, but they rarely disclose the exact SAE version, such as whether they support the updated `H2E` variant. This ambiguity complicates AP selection, requiring a balance between testing coverage across diverse devices and avoiding redundant testing of APs that only implement outdated or known-vulnerable versions. As a result, purchasing APs for fuzzing becomes a gamble, and SAECRED ends up meaningfully exercising only `3` out of the `6` COTS APs.

Lack of AP configuration controls. We report two COTS APs vulnerable to the downgrade attack, but the third AP—despite supporting `H2E`—only enables a single ECC group. Since most APs do not expose configuration options for selecting ECC groups, we cannot verify whether this AP is immune to the downgrade parsing bug or simply untestable due to its limited configuration. Consequently, SAECRED presents a conservative lower bound on the number of parsing bugs detected.

12. Conclusion

We presented SAECRED and exposed longstanding inadequacies in both Wi-Fi and fuzzing, leading to the discovery of parsing bugs missed for over four years. SAECRED demonstrates the need for context-sensitivity, state-awareness, and parser testing in realistic settings. Despite having only black-box access, SAECRED uncovers buggy parser behavior and achieves broader code coverage due to its holistic design. In future work, we plan to evaluate the generalizability of SAECRED by applying it to other network protocols with complex packet formats.

References

[1] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *CCS*, 2017.

[2] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the dragonfly handshake of wpa3 and eap-pwd. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[3] Mathy Vanhoef. Fragment and forge: breaking Wi-Fi through frame aggregation and fragmentation. In *30th USENIX security symposium (USENIX Security 21)*, 2021.

[4] Zilin Shen, Imtiaz Karim, and Elisa Bertino. Segment-based formal verification of wifi fragmentation and power save mode. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, 2024.

[5] Omar Nakhila, Afraa Attiah, Yier Jin, and Cliff Changchun Zou. Parallel active dictionary attack on wpa2-psk wi-fi networks. *MILCOM 2015 - 2015 IEEE Military Communications Conference*, 2015.

[6] IEEE Std 802.11. Wireless LAN medium access control (MAC) and physical layer (PHY) spec, 2016.

[7] IEEE Std 802.11. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, 2020.

[8] Jouni Malinen. hostapd.

[9] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. Dragonblood is still leaking: Practical cache-based side-channel in the wild. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, 2020.

[10] Matheus E. Garbelini, Chundong Wang, and Sudipta Chattopadhyay. Greyhound: Directed greybox wi-fi fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[11] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering*, 27, 2022.

[12] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August. USENIX Association.

[13] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[14] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021.

[15] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *ICST 2020*.

[16] Syed Md Mukit Rashid, Tianwei Wu, Kai Tu, Abdullah Al Ishtiaq, Ridwanul Hasan Tanvir, Yilu Dong, Omar Chowdhury, and Syed Rafiul Hussain. State machine mutation-based testing framework for wireless communication protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24.

[17] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4g lte cellular devices. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, 2021.

[18] Hongjian Cao, Lin Huang, Shuwei Hu, Shangcheng Shi, and Yujia Liu. Owfuzz: Discovering wi-fi flaws in modern devices through over-the-air fuzzing. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '23. Association for Computing Machinery, 2023.

[19] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. Evolutionary grammar-based fuzzing. In Aldeida Aleti and Annibale Panichella, editors, *Search-Based Software Engineering*. Springer International Publishing, 2020.

[20] Gaganjeet Singh Reen and Christian Rossow. Dpifuzz: A differential fuzzing framework to detect dpi elusion strategies for quic. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening pandora's box through atfuzzer: dynamic analysis of at interface for android smartphones. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, 2019.

[22] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jiaguang Sun. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model. *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[23] Kyungtae Kim, Sungwoo Kim, Kevin R. B. Butler, Antonio Bianchi, Rick Kennell, and Dave (Jing) Tian. Fuzz the power: Dual-role state guided black-box fuzzing for USB power delivery. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association.

[24] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. Lteinspector: A systematic approach for adversarial testing of 4g lte. In *NDSS*, 2018.

[25] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19.

[26] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the untouchables: Dynamic security analysis of the lte control plane. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[27] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. DoLTEst: In-depth downlink negative testing framework for LTE devices. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, 2022. USENIX Association.

[28] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, Bellevue, WA, August 2012. USENIX Association.

[29] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019.

[30] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*.

[31] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. cvc 4 sy: smart and fast term enumeration for syntax-guided synthesis. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*. Springer, 2019.

[32] Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. The sygus language standard version 2.1. *arXiv preprint arXiv:2312.06001*, 2023.

[33] IEEE Std 802.11s. *Amendment 10: Mesh Networking*, 2011.

[34] Dan Harkins. Dragonfly Key Exchange. RFC 7664, November 2015.

[35] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2), 1983.

[36] Jouni Malinen. SAE anti-clogging token. Retrieved 13 October 2024 from https://mentor.ieee.org/802.11/dcn/19/11-19-2154-02-000m-sae-anti-clogging-token.docx, 2020.

[37] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, 2010.

[38] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[39] Jouni Malinen. Sae h2e and incomplete downgrade protection for group negotiation. https://w1.fi/security/2024-2/sae-h2h-and-incomplete-downgrade-protection-for-group-negotiation.txt.

[40] Jouni Malinen. hostapd commit log: Password identifier dos patch. https://w1.fi/cgit/hostap/commit/?id=90a3b4a91a5dff29b8e8431983aacfc7aad52381.

[41] MediaTek. Security acknowledgement cve-2025-20664. https://nvd.nist.gov/vuln/detail/CVE-2025-20664.

[42] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[43] Michal Zalewski. Afl (american fuzzy lop).

[44] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022.

[45] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. 2015.

[46] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. *IEEE Transactions on Information Forensics and Security*, 2022.

[47] Dongge Liu, Van-Thuan Pham, G. Ernst, Toby C. Murray, and Benjamin I. P. Rubinstein. State selection algorithms and their impact on the performance of stateful network protocol fuzzing. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

[48] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: packet sequence oriented fuzzing for protocol implementations. In *USENIX Security '23*, SEC '23.

[49] Matheus E. Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. BrakTooth: Causing havoc on bluetooth link manager via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[50] Haram Park, Carlos Nkuba Kayembe, Seunghoon Woo, and Heejo Lee. L2fuzz: Discovering bluetooth l2cap vulnerabilities using stateful fuzz testing. *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022.

[51] Fabian Bäumer, Marcus Brinkmann, Nurullah Erinola, Sven Hebrok, Nico Heitmann, Felix Lange, Marcel Maehren, Robert Merget, Niklas Niere, Maximilian Radoy, et al. Tls-attacker: A dynamic framework for analyzing tls implementations. *Proceedings of Cybersecurity Artifacts Competition and Impact Award (ACSAC'24)*, 2024.

[52] Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *2017 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper presents a new fuzzing approach, SAECRED, to evaluate black-box implementations for flaws in Wi-Fi packet parsing. SAECRED combines a state-exploration approach with a packet grammar-based mutation approach. The paper reports vulnerabilities from many products.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction
- Other

A.3. Reasons for Acceptance

1) SAECRED appears to be generalizable for testing similar stateful wireless protocols with complex packet structures, making it a significant contribution to protocol testing methodologies.
2) The paper presents a well-designed evaluation framework, including ablation experiments, to demonstrate the effectiveness of its mutations, inter-field dependency representation, and state-aware methodology.

A.4. Noteworthy Concerns

1) The core methodologies (state-aware fuzzing, context-sensitive grammars, Iterative Deepening Search) are not new to the community. The novelty primarily lies in the application rather than a foundational advancement in security research.