Automated Analysis of Security Protocol Implementations

By

Christopher McMahon Stone

A thesis submitted to
the University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

Centre for Cyber Security and Privacy
School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
January 2021

ABSTRACT

Security protocols, or cryptographic protocols, are crucial to the functioning of today's technology-dependant society. They are a fundamental innovation, without which much of our online activity, mobile communication and even transport signalling would not be possible. The reason for their importance is simple, communication over shared or publicly accessible networks is vulnerable to interception, manipulation, and impersonation. It is the role of security protocols to prevent this, allowing for safe and secure communication.

Our reliance on these protocols for such critical tasks, means it is essential to engineer them with great care, just like we do with bridges or a safety-critical aircraft engine control system, for example. As with all types of engineering, there are two key elements to this process – design and implementation. In this thesis we produce techniques to analyse the latter. In particular, we develop automated tooling which helps to identify incorrect or vulnerable behaviour in the implementations of security protocols.

The techniques we present follow a theme of trying to infer as much as we can about the protocol logic implemented in a system, with as little access to it's inner-workings as possible. In general, we do this through observations of protocol messages on the network, executing the system, but treating it as a black-box. Within this particular framework, we design two new techniques – one which identifies a specific vulnerability in TLS/SSL, and another, more general approach, which systematically extracts a protocol behaviour model from protocols like the WiFi security handshakes. We then argue that this framework limits the potential of model extraction, and proceed to develop a solution to this problem by utilising grey-box insights. Our proposed approach, which we test on a variety of security protocols, represents a paradigm shift in the well established model-learning field.

Throughout this thesis, as well as presenting general results from testing the efficacy of our tools, we also present a number of vulnerabilities we discover in the process. This ranges from major banking apps vulnerable to Man-In-The-Middle attacks, to CVE-assigned ciphersuite downgrades in popular WiFi routers.

ACKNOWLEDGMENTS

First and foremost, I would like thank my main supervisor Tom Chothia. Tom has supported my journey into computer security research since I was a (clueless) second-year undergraduate. Over the 6 year period since then, he has fostered in me a uniquely relaxed approach to research, focusing on interesting and highly impactful topics. Our research sabbatical to the US was without question the highlight of my PhD. Many thanks also go to my secondary supervisor, Flavio Garcia, whose invaluable advice has been a great help in seeking past and future opportunities. His coffee machine, and the many accompanying social breaks surrounding it, were also essential in getting me to this point!

I have been very fortunate to share the PhD experience from start to finish with two now very good mates, Jan Van Den Herrewegen and Rajiv Singh. I look forward to a lifetime of friendship and reminiscences of these past years. Other special mentions go to my ever-supportive fellow Telfordian, Richard Thomas, as well as other friends and colleagues: Joeri de Ruiter, Andreea Radu, Kit Murdock and Nabi Omidvar. I would also like to sincerely thank Sam Thomas for our immensely productive collaboration in recent years. I am sure much more problem solving, stimulating discussions, and friendship awaits us.

On a more personal note, I want to thank my parents for their unwavering support over a seemingly interminable 21 years of full-time education. They thought they had got rid of me years ago, but the lockdowns of 2020/1 forced me upon them once again during my stressful write-up period. The last couple of years of my PhD were, however, made immeasurably more enjoyable by being spent with my girlfriend. You, Blanca, more than anyone, helped me get over the finish line.

Contents

Chapter One

Introduction

As global expansion of the internet began to rapidly increase, especially following the advent of the World-Wide-Web in the early 90s, so too did the demand for a means to protect networked data. Without data encryption, integrity and authentication, your e-mail, e-commerce and e-anything could be trivially eavesdropped, manipulated or blocked by anyone in your vicinity, and potentially done so with access to backbone internet infrastructure. This exposed an unacceptable risk for many internet operations and for computer networks more generally. Consequently, efforts began to design widely-deployable, software-based protection mechanisms known as *security protocols*. These protocols define standardised sequences of message exchanges and cryptographic computations which are carried out between two or more entities, and ultimately allow for subsequent secure communication. Today, such protocols are used, often unconsciously, by billions of people every day. Not only do these protocols protect our online activity but also our mobile communications, such as calls or texts, transport, such as train signalling and autonomous vehicles, and even industrial processes, from power plants to car manufacturing.

Despite their successful widespread adoption, the design and implementation of such protocols has been, and remains, a challenging problem in both science and engineering. One of the pioneers of security protocols, the late Roger Needham, who in 1978 designed the Needham-Schroeder protocol, framed the challenge aptly: "Crypto protocols are three-line programs that people still manage to get wrong". Fast forward nearly half a century, and issues are still occasionally uncovered in protocol design, and implementation flaws are commonplace. The widespread usage of security protocols today means these issues can have devastating consequences. The infamous OpenSSL Heartbleed vulnerability of 2014, which affected up to two thirds of the world's servers hosting secure web pages, was described in Forbes as "The worst vulnerability found since commercial traffic began to flow on the Internet" [112]. The flaw, which allowed for the leaking of arbitrary data in memory, such as passwords or encryption keys, was reportedly known about and actively exploited by the NSA for up to 2 years [102]. Although now fixed, this serves as just one of many examples of serious issues which highlight the attack surface security protocols present, and consequently the importance of engineering them with great care.

TLS (previously SSL), which is implemented in the above-mentioned OpenSSL and primarily used to secure HTTP websites, is certainly one of the most widely known and used security protocols of today. Many other security protocols have also been deployed, each adapted to their different environments and use cases—WPA/2 and WPA-Enterprise in WiFi Networks, SSH for remote shell management, AKA in telephony 4G/LTE, just to name a few. The varying protocols are not the only aspect of diversity however. Each protocol in turn has many different *implementations*, developed for a variety of platforms, and obviously done so by a large number of individual developers. To support this, standardised specifications for protocols are drawn up and serve as a basis to ensure interoperability between implementations. These specification documents constitute hundreds, if not thousands of pages (e. g. the 3000+ page IEEE 802.11 specification for WPA/2 [60]), often including subjective choices to made by developers. This, combined with the propensity for human error in both specification interpretation and coding, introduces inconsistencies between implementations. At best, this can manifest as a characteristic quirk, at worst, a security-compromising vulnerability.

In this thesis, we develop techniques to ensure *security* protocols live up to their name by helping to uncover some of the aforementioned issues. In particular, we consider the problem of analysing the *logic* which is implemented in security protocol software. At a high level, this means verifying that protocol messages are handled as they should be, irrespective of whether messages are received out of order or with unexpected contents, for example. The approaches we take are ones which aim to maximise *automation*, and abstract the complex inner-workings of protocol implementations. By doing so, we facilitate straightforward human analysis, requiring only an understanding of expected protocol functionality. This contrasts with the expert knowledge required to analyse the source code of implementations, which is a lengthy process, liable to oversights and mistakes, and often not even possible due to the proprietary, closed-source nature of some software. To this end, the protocol analysis approaches we present in this thesis are designed to infer information about the underlying implementations without access to source code. In general, we do this by a largely *dynamic* based analysis, extracting information from live observations of inputs and outputs (i. e. messages) to and from the target system. Compared with static code analysis, this approach provides greater assurances that any uncovered behaviour can actually be executed. Lastly, we ensure that this inferred information is clearly presented to an analyst, be that in the form of a notification that a particular (un)expected behaviour is present, or a more systematic behaviour *model*.

This thesis consists of three distinct units of research, each of which explore different approaches in order to meet the goals laid out above. We select two major, widely-used security protocols as cases studies—Transport Layer Security (TLS) and Wireless Protected Access (WPA/2), and examine a number of implementations of the two. We begin by considering the problem of identifying insufficient TLS certificate verification. In particular, we design a tool which can detect such issues even in the presence of so-called certificate pinning. Previous automated techniques would fail identify flaws involving this relatively recent technology, and manual approaches were limited due to their inability to scale. We test hundreds of high-security mobile applications, such as banking apps, including some from the major banks HSBC and Bank of America, and find a number of critical vulnerabilities in them. We report our findings, which results in fixes rolled out by developers and consequently improving security for millions of people. We offer in-depth analysis of how the vulnerability occurs and discuss remediation options.

Following on from this *black-box* approach for identifying a specific vulnerability, in the next unit, we explore how the same type of information can be used to more generally model the implemented behaviour of a protocol. Specifically, we adopt automata learning, whereby Mealy machine models are built through a series of input-output queries. The idea here is to systematically explore how an implementation handles protocol messages, irrespective of their ordering, and model the *states* of protocol execution. Whilst a well-established technique, we are required to enhance existing algorithms further, in order to handle unreliable communication and factor in time-related behaviour efficiently. Our resulting tool is tested on implementations of the WPA/2 security handshake, helping us to identify a number of vulnerabilities, including CVE-2018-0412 in some Cisco WiFi routers.

In the final unit of research, we explore what can be achieved with a relaxing of the black-box analysis requirement. In particular, we seek to utilise run-time introspection in order to aid automata learning of implemented protocol behaviour. Our devised approach represents a significant re-think of previous protocol model learning algorithms, instead classifying protocol states based on *memory snapshots* combined with input-output information. Experimenting on a series of both TLS and WPA/2 implementations, we find the new approach speeds up learning across the board. In some cases, this speed up is profound – resulting in an expansion in the type of protocol states that can be identified. This is especially the case for what we call *deep* states, which require long sequences of inputs to reach and are typically missed by black-box learning algorithms. We additionally present CVE-2020-17497, found by our tool in the Intel Wireless WPA/2 daemon.

1.1 Overview

We structure the thesis as follows:

Chapter 2 We begin by covering the required technical background on security analysis of protocol implementations. We discuss the predominant protocols that are considered throughout the thesis (TLS and WPA/2) and the fundamentals related to the analysis techniques we deploy, including model learning and binary program analysis.

Chapter 3 Building on the background of the previous chapter, here we provide a review of the related literature and state-of-the-art. This focuses on two key areas: implementation security analyses of TLS and WPA/2, and a critique of model learning learning algorithms which motivate the work of Chapters 5 and 6. We also cover related research on protocol reverse engineering and fuzzing – techniques from which inspire elements of the novel analyses and tools we present in this thesis.

Chapter 4 Here we introduce one of the central themes of this thesis, that of inferring implemented protocol behaviour from network I/O observations. The chapter covers our work on addressing deficiencies in existing automated techniques for detecting vulnerabilities in TLS server authentication implementations. In particular, we design an semi-automated tool which identifies flawed TLS certificate verification even if certificate pinning is in use. We test the tool on hundreds of mobile apps and present our findings. This chapter is based on the following publication:

McMahon Stone, C., Chothia, T., and Garcia, F. D. Spinner: Semi-Automatic Detection of Pinning Without Hostname Verification. In *Proceedings of the 33nd Annual Conference on Computer Security Applications* (2017), ACM

Chapter 5 Based on the theme of the previous chapter, here we take a broader look at how flaws in protocol logic can be identified. To this end, we adopt black-box automata learning in order to infer behaviour models of protocols. Previous studies demonstrate this to be effective for uncovering vulnerabilities in security protocols, however, our attempts to apply the method to the security handshake used in WiFi networks reveal a number of shortcomings. We consequently design and implement improvements to the state-of-the-art, test their effectiveness through application to a number of WiFi access points, and present the vulnerabilities we discover. This chapter is based on the following publication:

McMahon Stone, C., Chothia, T., and de Ruiter, J. Extending automated protocol state learning for the 802.11 4-way handshake. In *European Symposium on Research in Computer Security* (2018), Springer, pp. 325–345

Chapter 6 In this chapter we discuss the inherent limitations in black-box learning which motivate us to investigate alternatives. We proceed by easing the black-box restriction of the technique, and explore how dynamic run-time memory inspection and binary program analysis can aid learning. We devise a novel *grey-box* approach that classifies states of the protocol via memory snapshots. We locate individual bytes of memory which maintain program state through advanced snapshot "diffing" and a hybrid taint-symbolic execution based analysis. We go on to demonstrate how this enables us to learn more accurate models than black-box learning and the potential for this work to facilitate the application of further analysis techniques. This chapter is based on the following publication:

McMahon Stone, C., Thomas, S. L., Vanhoef, M., and Chothia, T. Grey-box Protocol State Machine Learning. In *Under Submission*

Chapter 7 Here we conclude the thesis, providing reflections on the work of the previous chapters, and discuss directions for future work.

Part I

Background, Preliminaries & Related Work

Chapter Two

Background & Preliminaries

In this thesis, we concern ourselves with security testing of protocols, specifically implementations of cryptographic network protocols. This chapter lays out the required background related to our analysis of such protocols. In particular, we first provide an overview of both Transport Layer Security (SSL/TLS), and Wi-Fi Protected Access (WPA/2). We then proceed to cover material related to the analysis methods we build upon and apply to these protocols. This includes, state machine (a.k.a model) learning, and the binary analysis methods of dynamic taint tracking and symbolic execution.

2.1 Security Protocols

TLS and WPA/2 are two of the most widely used security protocols. Both protocols are deployed in order to carry out two key functionalities – channel confidentiality and integrity, and authentication of one or more of the parties involved. In this section, we provide a high-level overview of these protocols, with additional focus on features relevant to to the analyses we carry out.

2.1.1 Transport Layer Security (TLS)

Overview

The TLS protocol is the successor to Secure Sockets Layer (SSL) which was first released in the mid-nineties. Since this time, the protocol has undergone a number of significant redesigns due security analysis efforts from researchers and practitioners across the world. At the time of writing, TLS 1.2 is the most widely deployed version of TLS. However, the more recent TLS 1.3 is seeing increasing adoption due to its improved speed and security. In this thesis we focus on analysis of TLS 1.2 implementations.

In summary, the main key-negotiation and authentication handshake for TLS 1.2 begins with a client indicating its desire to connect to a server, along with a selection of supported cipher suites (i. e. key exchange, encryption and MAC computation algorithms). The key exchange is then carried out and a session key is calculated in combination with a random nonce contribution from both parties. During this process, the server will present a certificate to the client which is verified for authentication purposes. Optionally, the client can also be authenticated by the server through a similar certificate exchange. After the key exchange and authentication, both parties switch to an encrypted channel, where a final message is exchanged to verify the integrity of the whole handshake, and if successful, indicate the handshake is complete.

Authenticating Servers

In order to authenticate TLS servers, part of the initial handshake involves the communication of a server's X.509 certificate. The purpose of this certificate is to bind the server's identity to a given public key. It is the job of the client to verify this certificate so that it can be sure the public-private key pair belongs to the server it intends to be communicating with. Once this verification process is complete, the public key encapsulated in the certificate is used to negotiate a session key.

In a typical TLS set-up, this trust is enabled through the maintenance of a set of by-default trusted root Certificate Authority (CA) certificates. Servers wishing to use TLS must obtain their own cryptographically signed certificate from one of these CAs. A client wishing to set up a TLS connection will then validate the server's certificate by carrying out a number of checks. This includes: recursively checking the signatures of each certificate in a chain to a trusted anchor, verify the certificate was issued to the expected host, and ensuring the certificate has not been revoked or has expired.

Unfortunately, Certificate Authorities are not immune from compromise, which if were to happen (like with DigiNotar [51] in 2012), would enable the perpetrator to generate valid certificates for domains of their choosing. These forged certificates could then be use to Man-in-the-Middle TLS connections from any client trusting the compromised CA. Furthermore, a number of CAs have been caught mis-issuing certificates. In early 2017, Google announced plans to gradually phase out trust of Symantec certificates in the Chrome browser, citing concerns that the organization had mis-issued thousands of certificates [108]. Developers are further motivated to avoid reliance on a devices trust store due to the potential for users to be phished into inserting malicious certificates into to trust store.

Certificate Pinning

In many mobile applications, unlike generic browsers, the client knows the identity of the server(s) in advance. The consequence of this is that dependence on CAs can be entirely removed, or reduced down to a single CA. Conceptually, certificate pinning achieves this by allowing a developer to implement their own trust store. In practice, certain elements of a certificate chain are fixed or hard-coded into the application. Exactly which certificate in a chain is pinned depends on the developer's requirements:

- **Leaf Certificate** - The most obvious element to pin is the end-entity or leaf certificate. No CAs are relied upon and attack surface is minimal. Flexibility is reduced however; if key rotation policy mandates that the certificate is changed, then users will be required to update their apps to continue use, similarly when the certificate expires.

- **Intermediate Certificate** - Certificate chains will typically be of length 3 or more. If the certificate is neither a root, nor a leaf, then it is called intermediate. Pinning to this type of certificate enables the end-entity to easily renew it's leaf certificate (as long as it is signed by the pinned intermediate).

- **Root Certificate** - Pinning to a root certificate is the most flexible approach as length of validity is usually $\sim$20 years.

Developers must also deliberate over whether the entire certificate or just the public key is pinned. Pinning the certificate is the most common implementation, but suffers from issues regarding certificate expiration. On the other hand, public key pinning is less widely used, but allows for key continuity even when the certificate has expired.

Server Name Indication

In standard TLS, it is a requirement that each host, with its corresponding certificate, has its own IP address. The implication of this is that virtual hosting, where multiple domains, each with their own certificate, share a single IP address, is not possible. To overcome this limitation, the Server Name Indication (SNI) extension was proposed in RFC6066 [47]. The solution works by enabling the client to specify the host it wishes to communicate with in the ClientHello message. The server can then inspect this value and serve to the client the appropriate certificate for the requested hostname.

In the research of Chapter 4, we use SNI to facilitate transparent TLS proxying.

As some mobile applications bypass manual proxy settings, we require a methodology that transparently intercepts and redirects network traffic. Unfortunately, DNS spoofing on its own does not suffice. Many clients will make numerous DNS lookups before making any TLS connections. Therefore we use SNI to distinguish between TLS connections, so that they can then be proxied to the appropriate host.

2.1.2 Wireless Protected Acess (WPA/2)

All secured Wi-Fi networks are protected with some version of Wireless Protected Access (WPA). In 2004, the 802.11i amendment to the original 802.11 standard was released and implemented in WPA/2. This standard defines a 4-Way Handshake in order to carry out session key negotiation and authentication. Any full, WPA/2-certified network connection will however begin with a network discovery phase and the 802.11 authentication[1] and association stage. Each of these stages is described below.

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

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

[1]In name only, remains largely for backward compatibility purposes.

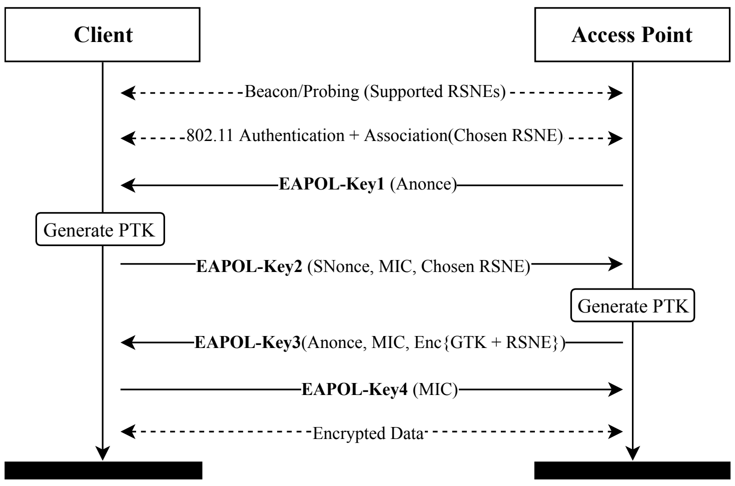

Client
Access Point
Beacon/Probing (Supported RSNEs)
802.11 Authentication + Association(Chosen RSNE)
EAPOL-Key1 (Anonce)
Generate PTK
EAPOL-Key2 (SNonce, MIC, Chosen RSNE)
Generate PTK
EAPOL-Key3(Anonce, MIC, Enc{GTK + RSNE})
EAPOL-Key4 (MIC)
Encrypted Data

Figure 2.1: The WPA/2 4-Way Handshake.

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

The 4-Way Handshake, as shown in Figure 2.1, is initiated by the AP, who communicates its nonce to the client. The client then generates its own nonce, and sends it to the AP in Message 2, along with a Message Integrity Code (MIC) that is calculated over the whole frame using the PTK. The AP can then verify the client has derived the correct PTK by generating the PTK itself, and checking that the MIC is valid. It can also detect a downgrade attack by verifying that the RSNE matches that in the earlier Association stage. If all is well, the AP responds with Message 3, which contains the encrypted Group Key and RSNE. The client can then verify the RSNE is consistent with previous messages, if so acknowledge with Message 4 and if not, abort the connection.

Messages are encapsulated within EAPOL-Key frames. These include nonces, version numbers, MICs, replay counters and so on. In our model learning of the 4-Way Handshake in Chapter 5, we only consider the most crucial of these (with respect to security). The reader can find complete information on EAPOL-Key frame structure and contents by referring to the 802.11 specification [60].

2.2 Model Learning

In recent years, model learning has emerged as an effective technique for learning black-box systems. Extracting models of such systems has proved effective for a wide range of applications, including: aiding model-based testing [52, 114], security protocol fuzzing [44], GUI testing [35], learning legacy systems [79], and aiding symbolic execution [8], just to name a few.

The classical procedure for learning a system is L^* (or Angluin's algorithm) [12]. Here, the System Under Learning (SUL) is considered to be a regular language, whose alphabet is known. The algorithm constructs successive queries consisting of words from the alphabet, and a Deterministic Finite Automata (DFA) is gradually inferred. The resulting DFA approximates the acceptor for the language.

In order to model complex software systems however, like the protocols implementations we consider in this thesis, other types of automata are preferred. This is because the behaviour of such systems is typically characterized in terms of I/O. Concretely, in a given state of a protocol execution, the system will accept an input, carry out some computation, produce an output and then transition to a different state. To model this, Mealy Machines are considered the most appropriate. Although DFAs can be used, Niese demonstrated that Mealy Machines are capable of representing the same system in far fewer states than a standard DFA [89].

Mealy Machines

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

Less formally, when a Mealy machine is in a state $q \in Q$ and receives as input $i \in I$, it transitions to a target state defined by $\delta(q, i)$ and produces an output corresponding to $\lambda(q, i)$.

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

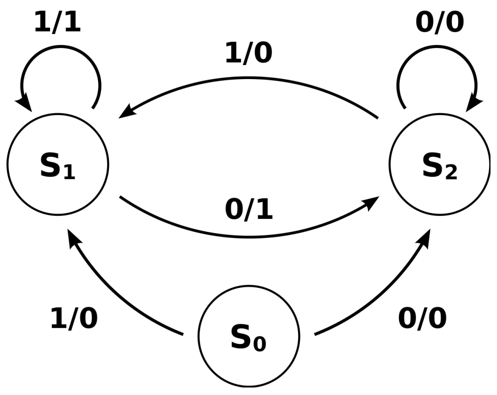


Figure 2.2: Basic Mealy machine with $I = O = \{0, 1\}$ and $Q = \{s_0, s_1, s_2\}$

Mealy Machines Learning Overview

An adaptation of Angluin's L^* algorithm for learning Mealy machines was first conceived by Niese [89] and later optimized by Shabaz et al. [105]. Like L^*, the solution is modelled such that there are two main components: An *oracle* (or *teacher*), that acts as an interface to the executing SUL, and a *learner*, that is only aware of the input and output symbol sets I and O, and can additionally request the oracle to reset the SUL to the start state

q_0.

The algorithm proceeds by asking *output queries* that are strings from I^+. The oracle responds with the corresponding output strings from the machine. Each *output query* is preceded by a *reset query*.

The queries are asked iteratively and responses are recorded in an observation table. This continues until the table has been sufficiently populated so that particular conditions have been met. Once done, the recorded data forms the basis for a Mealy machine conjecture.

The next stage of the algorithm is to ask equivalence queries to the *oracle*. These are of the same format as output queries, however, are made with the aim of finding a difference between the conjecture and the actual Mealy machine. If the *oracle* states that the conjecture is correct, the algorithm terminates. Otherwise, the oracle will respond with the contradicting output string, i.e. a *counterexample*. In the latter case, the *counterexample* is processed along with the table so that the conjecture can be refined.

Equivalence queries can be generated in a number of different ways. One such example is Chow's W-method [38], which can guarantee that, given an upper bound for the number of states, the correct FSM can be always be found. Alternatively, if a more efficient but less complete solution is desired, equivalence queries can be generated randomly. Note that as this is traditionally a form of black-box testing, i.e. the oracle cannot access the internal implementation of the SUL, and only a finite number of test cases can be performed, the equivalence checking can only be approximated. For example, if the actual implemented Mealy machine changed behaviour after very large number of specific output queries are made, then it is unlikely to learn this in polynomial time. The consequence of this is that in some cases, it may only be possible to learn a subset of the SUL's behaviour with black-box testing. We take steps towards addressing this limitation in the research of Chapter 6.

2.3 Binary Program Analysis

In this section we provide an overview of the binary program analysis techniques deployed in the research presented in Chapter 6. In particular, we utilise a combination of Dynamic Taint Analysis and Symbolic Execution to construct a novel analysis that aids model learning of protocol implementations. These two, well studied techniques, have previously been sucessfully used for a variety of purposes, including, but not limited to, vulnerability discovery [113, 123], test case generation [104, 29] and malware analysis [16, 15].

2.3.1 Dynamic Taint Tracking

At a high-level, taint propagation tracks the influence of a given location (i. e. memory buffer or register) along an execution path. Given a location we wish to track, a taint *seed* can be introduced by assigning each byte or bit of the location a taint tag. For each program instruction that operates on a tainted bit or byte, the associated taint tag is propagated to any location (at the byte or bit granularity) that is computed, or written to as a result of that operation. This concept can then be extended to account for control-dependence by propagating taint tags of locations that taint conditional operations that subsequently decide if a branch is taken to the instructions executed after the branch. Finally, a taint tag may be removed from a location if the location is processed by a *sanitisation* operation: this might for example be when the tainted buffer is overwritten with a constant, e.g., via `memset`. At any point during analysis, if a location or instruction is marked with a given taint tag, then its value (in the case of memory) or execution (in the case of an instruction) will have been influenced by the taint source that introduced the tag.

2.3.2 Symbolic Execution

Similar to taint propagation, symbolic execution traces properties of a program location along a given execution path. Instead of tracking dependence, symbolic execution tracks constraints over the possible values the location can store. For each instruction that operates directly on a symbolic location, or a value derived from it, constraints are built from those instructions to express properties of the location, treating it as a bit-vector. Using these constraints as input, an SMT solver (e. g. Z3 [43]) can be queried to provide a model (i. e. a viable assignment of the symbolic location) satisfying the constraints. We may negate one or more of these constraints to learn assignments that trigger different program behaviours.

Chapter Three

Related & Previous Work

As implied by the content of Chapter 2, analysing the security of software like a protocol implementation draws on techniques and algorithms from a number of distinct fields. In this chapter, we explore the existing literature related to those fields.

We begin by investigating literature specific to security analysis of the protocols considered in this thesis. This consists of a wide body of work, which, aside from key techniques we build upon, includes: manual based analyses, static analysis, fuzzing, and more. We then focus on the primary methodology used in Chapters 4 and 5 of this thesis – model learning. As our research is concerned with improving model learning algorithms, we provide a review of the developments and key research into these underlying algorithms. We also look at various studies which, like our work, use these algorithms for learning models of protocol implementations.

A closely related field to security testing of protocols is that of protocol reverse engineering. Although this is primarily concerned with identifying the messages and behaviour of e. g. proprietary or malware protocols, the approaches taken overlap substantially with those used in this thesis. This is similarly the case for protocol fuzzing, and especially so in the context of stateful protocol fuzzing. For this reason, we additionally provide a review of key works stemming from these two fields which help to guide our thinking in designing the new approaches presented in this thesis.

3.1 Security Protocols

3.1.1 TLS

General

TLS is the defacto standard for securing internet traffic. Originally proposed as the Secure Sockets Layer (SSL) by Netscape in 1994, the protocol has experienced significant evolution over the years. The first two versions of SSL were seriously flawed and barely made it off the ground (indeed, SSLv1 was never publicly released). SSLv3 and the subsequent TLS 1.0, although more serious contenders, were also found to be flawed in many ways, including: padding oracle attacks like POODLE [87], predictable IVs in the BEAST (CVE-2011-3389) attack, just to name a couple. However, as the TLS specification matured, especially with TLS version 1.2, the last 10 years have seen research efforts focus more on the testing of individual implementations.

In 2015, two groups of researchers took two separate approaches for analysing state machines in TLS implementations. Beaurdouch et al. use the FLEX [18] TLS testing tool in a model-based testing approach [17]. The authors carefully produce a model of the protocol and generate various tests to check deviations from said model. In [44], de Ruiter et al. instead deploy model learning in an approach that forms a large part of this thesis. In comparison to [17], model learning allows for a more automated style of analysis. As tests are automatically generated by the model building process, greater coverage can be achieved, allowing for detection of a larger class of erroneous or vulnerable behaviour. We refer to de Ruiter et al.'s model learning work further in Section 3.2.2, and later in the contributing chapters.

In 2016 Somorovsky et al. [110] introduced a new tool for testing TLS implementations. The extensible framework implements a large proportion of TLS functionality and is ideal for dynamic analysis techniques such as model learning and fuzzing. The tool has been used in a wide-array of implementation-security focused works on TLS, include fuzzing for memory corruption vulnerabilities, identification of padding oracles or Bleichenbacker attacks, and particularly relevant to this thesis, recent model learning of DTLS implementations [53].

One key aspect of TLS implementations that is hitherto unmentioned, yet has warranted much past research focus, is that of certificate verification. We discuss previous works on this notoriously complex part of TLS, particularly related to mobile apps and our work in Chapter 4, in the next section.

TLS Server Certificate Verification

There are a number of past studies that have looked at certificate verification in non-browser software. Fahl et al. [49] developed a tool that statically analyses Android code and detects non-standard implementations. They then go on to manually test for Android apps that accept self-signed certificates or do not verify hostnames. Carrying out this process with over 10,000 apps, they find a surprisingly large portion of them contain these serious flaws and hence expose themselves to Man-in-the-Middle attacks.

Georgiev et al. [57] independently carried out a similar study. In addition to Android, they also considered other platforms and applications such as instant messenger clients, merchant payment SDKs and cloud client APIs. Brubaker et al. additionally developed a tool to fuzz certificates [26]. By generating certificates with mutated fields, and then presenting them to clients during a TLS handshake, they also find many critical bugs in the verification code in a number of implementations.

Sounthiraraj et al. [111] improved on the analysis techniques proposed in [49].

They develop a tool named SMV-Hunter, which, in addition to statically analysing source code to detect non-standard implementations, carries out dynamic tests by automatically running apps and emulating intelligent user input to trigger TLS connection attempts. Basic TLS certificate mis-verification flaws are then detected automatically. We note however that this tool also lacks the capability to detect hostname verification flaws when certificate pinning is being used.

Concentrating on the hostname verification process, Sivakorn et al. [107] find that the set of acceptable hostnames specified by the patterns in the CommonName (CN) and SubjectAltName values form a regular language. They use automata learning algorithms to infer the corresponding models, and then use this to find discrepancies in particular hostname verification implementations. Running these tests on popular TLS libraries, they find 8 unique violations of RFC specifications, several of which render them vulnerable to Man-in-the-Middle attacks.

Previous research has also focused on applications implementing certificate pinning. Oltrogge et al. [90] scanned a large portion (around 600,000 apps) of the Play Store to find out how many were employing this additional security measure. Using various metrics, they determine and recommend that significantly more apps would benefit from certificate pinning. They also interview a number of developers and find that general understanding on the topic is good, but the complexity of implementation methods makes it too difficult to use. Implementations with serious flaws discovered in [37] backup this finding.

Previous work on automated tools has not considered the case of missing hostname verification when certificate pinning was being implemented. Chothia et al. [37] manually analysed the way in which TLS was being used in apps from the UK's largest banks. They found a number of misuses, including two apps from major banks that pinned a root CA certificate but did not validate hostnames. They tested for this by purchasing a certificate from the same CA being used by each app, but for a domain they owned. The research presented in Chapter 4 of this thesis provides an automated approach for identifying this vulnerability and removes the need to purchase certificates.

Most recently, late 2020 saw the release of yet another analysis of hostname verification implementations. In particular, GoSecure reported two critical vulnerabilities in Oracle HostnameChecker and Apache HTTPClient [96]. The root cause of the flaws was the way in which character-set transformations were handled, allowing for hostname collisions. This serves to highlight the continued relevance of testing the implementations of this critical element of TLS security.

3.1.2 WPA/2

Wi-Fi drivers and the Wi-Fi security protocols (WEP, WPA, WPA/2, WPA-Enterprise) have been the subject of a wide array of past research and analysis. In 2001, Fluhrer et. al [55] discovered critical flaws in how the RC4 cipher is used by the original WiFi security mechanism, WEP. This enabled them to recover encryption keys and hence decrypt and inject network packets. This attack was made further practical by Beck and Tews [115], where the number of observed frames to recover the key was dramatically reduced—a final nail in the coffin for WEP. WEP was subsequently replaced by WPA, starting with WPA-TKIP, which was used in the interim period before WPA/2 (AES) could be rolled out. The 4-way handshake, which is deployed in WPA to authenticate clients and negotiate session keys, has undergone extensive formal analysis [62, 85, 63, 122]. In [62, 85], Denial of Service vulnerabilities were discovered. Here, authenticator messages are forged to induce inconsistent keys between the client and AP. The authors propose improvements and then give a correctness proof in [63]. These changes were then integrated into the 802.11i specification. The design of the 4-way handshake was further analysed by Vanhoef et al. [117], where the transmission of the group-key was a key focus. Here they find that it is vulnerable to a downgrade attack that forces the group key to be encrypted using the vulnerable RC4 cipher.

The security of WiFi implementations has also been the subject of many studies. Wi-Fi drivers have been tested using fuzzing methodologies. This helped Butti [28] and Mendoncca [84] detect numerous errors such as buffer overflows and Null pointer dereferences. Included in this were vulnerabilities which could be exploited to achieve remote compromise, privileged code execution and more. This work was however restricted to testing how unprotected management frames are handled. It was only until nearly a decade later, that implementations of the 4-way handshake were first tested.

Similar to the previously mentioned approach taken by Beurdouche et al. [17] on TLS, Vanhoef et al. [119], apply model-based testing in an effort to detect logical flaws in the 4-way handshake. The authors model the protocol by defining a sequence of messages exchanged in a normal handshake. Test cases are then generated from a set of rules that are applied to the model. These rules define a) when a test case should be executed (injecting frames before or after each normal message, or dropping messages) and b) the parameters of the injected messages (invalid cipher suite, invalid nonce etc.). Executing these tests on a number of APs, the authors discovered DoS and downgrade attacks, as well as fingerprinting mechanisms. As already explained however, this approach suffers from its lack of automation and coverage potential.

More recently, in their ongoing analysis of WPA, Vanhoef et al. discovered a series of vulnerabilities in how retransmissions of key exchange messages are handled. In their KRACK paper [118], the authors discuss how the keystreams used for encrypting data frames can be forcibly reused. The flaw, inherent in the 802.11i specification, makes clients reset the IVs used for encrypting packets in a given session. This enables an attacker to decrypt and replay in AES-CCM, and additionally recover the session key and forge messages if AES-TKIP (or GCMP) is used. This issue impacts the 4-way handshake, the Group key handshake and Fast-BSS transition. In Chapter 6, we show how automated model learning can be used to identify a variant of this exact flaw in one particular implementation.

Finally, there has also been some past work on security analysis of WPA-Enterprise. One study of particular relevance to our work is that of Brenza et al. [24]. Here, the authors identify a bug in the client state machine that manages the EAP-TTLS inner-authentication. This enables them to carry out Evil Twin attacks (see [58]), where clients are tricked into connecting to an AP in the control of an adversary. The exploit works because clients were found to be not properly checking the "Authenticator Response", sent at the end of a handshake. The authors discover this flaw through code inspection of the open-source Linux implementation `wpa_supplicant` and `hostapd`. With our model learning techniques adopted in this thesis, these type of flaws can instead be detected automatically. This saves time whilst also making it much easier to test closed-source implementations.

3.2 Model Learning

3.2.1 Fundamentals

Mealy Machines

In her seminal paper from the 1980s, Angluin presented the first algorithm for inferring automata of regular language acceptors [12]. This work was then later adapted by Niese [89] and further by Shabaz et al. [105], for learning the more expressive type of automata—Mealy machines. The prevailing theme in ensuing studies on Mealy machine inference consisted of optimizations and improvements in the processing of counterexamples. In [103], Rivest et al. show how the counterexamples produced by the naive method in [12] are not minimal and therefore affect the algorithmic complexity negatively. They go on to propose improved variants of L^* which are then further refined by Maler [78] and adapted for Mealy machine inference in [105, 67, 66].

One of the most significant developments of Mealy machine inference theory in recent years is the work of Isberner et al. [68]. Motivated by the same issue of inefficient handling of redundancies within counterexamples, the authors completely redesign L^* with the use of Pushdown automata. They achieve a totally redundancy-free algorithm that is the most efficient to date. Clearly, the continually improving algorithms for black box model learning mean that it is preferable to make any proposed adaptations independent of the learning algorithm. We ensure this is the case in the work we present in Chapter 5 for learning Mealy Machines with time-behaviour and non-deterministic learning. To do so, we take inspiration from transducer-based approaches taken by Aarts et al. [3, 7], which we look at in more detail in the proceeding sections.

An additional noteworthy development of late is the work of Groz et al. [61, 23] on learning systems without the ability to reset (see Mealy machine learning requirements in Background Section 2.2). In particular, they argue that for some Mealy machine systems, full state resets are expensive (e. g. embedded systems) or even not possible (e. g. remote inference). The proposed approach, which utilises so-called *homing sequences* [103], proves to be effective, but only for highly connected systems. This, however, is unfortunately not the case for typically sequential security protocol state machines.

The grey-box approach we propose in Chapter 6 take a radically different approach in solving some of the problems tackled in the previously mentioned works. In particular, our instant state identification through memory snapshots allows for a significant reduction in queries (meaning less system resets) and ability to explore *deeper* states due to more targetted state exploration (meaning less expensive search for counterexamples).

Timed Automata

A common feature of many software and hardware systems is that of time dependent behaviour. For example, in a protocol handshake a certain output may only occur if some input has not been provided within a particular amount of time (e. g. a timeout). The model learning algorithms described previously are generally not capable of learning this class of behaviour. As we will show in Chapter 5, this is an unfortunate limitation as critical behaviour (or even vulnerabilities) require consideration of time in order to be identified.

A number of past works have looked at the possibility of learning automata that capture time, formally known as Timed Automata [9]. This includes Grinchtein et al. [59] who propose an active learning approach for one-clock per action automata. The resulting algorithm however is known to be of prohibitively high complexity, and as such no implementations exist due to its impracticality. Later on, Verwer et al. [120], examined the algorithmic complexity of learning timed automata with multiple clocks and determined efficient learning to not be possible. Informed by these studies, in Chapter 5 we focus on practicality and make realistic assumptions on the types of time behaviour that may exist in the context of learning security protocol models. We describe similar practical approaches for handling time in this context in the proceeding discussion of Section 3.2.2.

We note that post-publication of our work [81] (Chapter 5), a pre-publication on learning Mealy machine with timers by Jonsson et al. was made available online [72]. The algorithm presented makes significant advances in the theory of learning systems with timers, however no implementation is yet available.

I/O Automata

Of particular relevance to our approach in Chapter 5 is that of Aarts et al. [7] on learning I/O automata [77, 71]. These automata are very similar to Mealy machines, however remove the restriction that inputs and outputs have to alternate. To learn the systems described by this automata variant, the authors introduce the concept of a Learning Purpose (LP). The LP is implemented as a transducer, positioned between the Mealy machine learner and teacher (i. e. the SUL). The function of the LP is to dictate to the learner when a given output query is allowed (e. g. only when the SUL becomes quiescent after the previous query, or each input is followed by at most one output, etc). This enables the learner to focus on learning particular parts of the system.

In order to learn time and retransmission behaviour in implementations of the WiFi 4-Way Handshake, in Chapter 5 we adopt the aforementioned approach and utilise a learning purpose to retrofit Mealy machine learning algorithms for such systems. We also show how the efficiency of this approach can be greatly increased with the use of caching.

Register Automata

One further way of modelling a system such as a protocol implementation is using the Register automata formalism [73, 30]. These automata have a finite set of states, but are extended with a set of *registers* that can be used to store data values. Input/output actions are then parametrised with values for these registers such that guard conditions can be enforced on transitions (e.g. equality). One such approach for learning these systems was implemented in the tool Tomte [3, 2]. The idea is to start with a drastic abstraction where data values are mostly ignored in I/O messages. The learner will observe non-determinism when the abstraction is too coarse and will use this information to gradually concretise specific data values. Although a promising development in expanding the capability of model learning, this approach is currently only able to learn academic examples e. g. the bounded retransmission protocol [5] or simple data structures. Advancements in algorithms for learning more complex Register automata will likely require approaches that utilise information beyond that available in black-box learning. In Chapter 6 we present a grey-box approach which similarly aims to address fundamental limitations in black-box learning.

3.2.2 Applications

In recent years, particularly upon release of the LearnLib library [99, 69], automata learning has been successfully applied in numerous domains. There have been applications in industrial integration testing, [106], automotive component testing [50], legacy software demystification [79], stateful network function learning [88], and combined with fuzzing for software deobfuscation [70].

More relevant to our work, however, are the applications to network protocols and security protocols. First and foremost, in [44], de Ruiter et al. analyse client and server implementations of the TLS security protocol, in a process they name *protocol state fuzzing.* The authors implement a test harness, similar to that of TLS-Attacker [110], which supports TLS message construction and parsing for key functionality, including several key exchange algorithms and client certificate authentication. Multiple critical vulnerabilities are discovered in a wide range of implementations, epitomising the effectiveness of this approach, and inspiring much of work in this thesis. In both Chapters 5 and 6, we identify key limitations of this original approach, including handling non-determinism, time behaviour, and the restrictions of a black-box based approach.

Not in a security setting, but still bearing relevance due to their handling of timing and retransmissions, Fiterău et al. [52] carry out a combined model learning and model checking of TCP implementations. They state that due being restricted by the lack of expressivity of Mealy machines, they have to eliminate the timing-based behaviour and retransmissions. The former was handled by verifying that learning queries were short enough so as to not trigger any timed behaviour, and for the latter, they ensured that the network adapter ignored all retransmissions.

Similarly, in a study involving the application of active learning to IoT communication, Tappler et al. [114] also note how they deal with timeouts. They adopt the technique used by [44], whereby a timeout is set for the receipt of all messages to a query.

All messages received within that time are then mapped to an abstract output symbol. The problem with this approach is that it does not allow queries that are interleaved between consecutive message responses. On the other hand, our approach, presented in Chapter 5, allows queries in between consecutive output messages if they are different (i. e. not between retransmitted messages).

Implementations of another widely-used security protocol, SSH, have also been analysed in previous works [54]. Like in our analysis of WiFi in Chapter 5, efforts are made to handle non-determinism. In particular, all observations made during learning are recorded in a database. This way, all queries to the SUL are checked for consistency with previous ones. If a query was flagged as inconsistent, the authors identify the cause manually and the learning configuration is adjusted accordingly. A similar manual-based handling is taken in efforts to learn OpenVPN [42]. Conversely, our approach removes this manual element and replaces it with fully automated error-correction. This also plays a role in facilitating reliable learning of time-based behaviour, which these previous studies also avoided.

Other studies of note include an analysis of the SIP protocol [4], the biometric passport [6] and bank cards [1].

3.3 Protocol Reverse Engineering & Fuzzing

A key related field to protocol implementation security testing is that of protocol reverse engineering (PRE). This field tackles two distinct challenges—protocol message set (or language) inference, and secondly, inferring the grammar which dictates the ordering of messages in protocol exchanges. As the contributions of this thesis are concerned more with testing the *logic* implemented in protocols, we focus on the latter. Grammar inference in the context of PRE can be broadly split up into two approaches [45], Network based, and Application based, each of which are covered in the next two sections.

In addition to reverse engineering, in this section we also cover relevant works in the domain of fuzzing, which again share many of the same techniques, however are largely focused on the traditional fuzzing goal of uncovering memory corruption style bugs and vulnerabilities.

3.3.1 Network Based Grammar Inference

The challenge of inferring the grammar of a protocol from the perspective of the *network* is very much like the black-box model learning work discussed in the previous Section 3.2. Indeed, the same approach was deployed in the Netzob tool of Bossert et al. [21, 22], which was used to reverse engineer and model botnet protocols. In particular, Netzob utilises L* Mealy machine inference, with adaptations for denoting transitions with probabilities and time. One particular interesting aspect of this work was the decomposition of target protocols into a series of sub-automata representing distinct functionality, so that learning complexity can be reduced. Compared to Botnet protocols, the security protocols in this thesis generally cannot be split up as such.

Other studies in this domain primarily adopt alternative algorithms to automata learning. The tool Veritas takes a probabilistic approach [124]. Messages are observed from traces and are grouped by the most frequent sequences in headers. The algorithm then builds a state machine assuming messages only depend on their previous counterparts, and then assign probabilities to which message is expected at each state. As the method is passive, the coverage is inherently limited. Additionally, any "unexpected" (i. e. low probability) observations in traces are not included.

Other successful approaches, like ASAP [75] and later PRISMA [74], are largely too domain specific to bear much similarity with the work of this thesis. PRISMA adopts a Markov chain modelling formalism which is the same modelling format adopted by Pulsar [56], a highly relevant grey-box technique which we discuss further in related work on fuzzing, Section 3.3.3.

3.3.2 Application Based Grammar Inference

Inferring the grammar, or state machine, of a protocol using application based insights means to *open* the black-box. The means to either inspect source code (white-box) or binary code (grey-box), analyse live execution traces, or a combination of the both. Indeed, in Chapter 6, we do exactly this—a grey box style analysis by monitoring execution and analysing binary code with taint tracking and symbolic execution. It is therefore important to cover key related works in this area.

One of the pioneering works in grey-box based PRE is that of Prospex by Comparetti et al. [40]. This work extends previous research by the authors [125] on extracting protocol message formats to include consideration of state machines. The approach consists of passively recording protocol sessions, and replaying them to trace execution. In each trace, the bytes of each message are tainted and the processing of individual messages is delimited via alternating writes to an output socket. Messages of the protocol are then categorised (i. e. abstracted) into equivalent groups through clustering. The clustering similarity metrics are broken down into two categories: ① Execution Similarity—consisting of a system call feature, invoked functions feature (including library/dynamically linked functions) and executed address feature, followed by ② Impact Similarity, which consists of an output feature (including writes to client socket, other sockets, files etc.), whether output is tainted, and a file system feature, which captures filesystem i/o. Using the learned message types, the goal then is to learn an acceptor machine which recognises sequences of message types which make a valid session. To this end, an APTA automata (c. f. Mealy machines used in this thesis) representing all sessions and constituent states as acceptor states is constructed by finding the smallest automata consistent with the training set. To make this minimal, state merging then takes place by learning input message pre/co-requisites (i. e. dependencies) using regexs. Any states with the same dependencies are merged.

In the description of our grey-box approach, StateInspector (Section 6.4.1), inspired by Prospex's delimitation of messages through alternating socket writes, we describe a similar method of choosing to snapshot state under the same assumption of message delimitation (i. e. at reads and writes to network sockets). Likewise, we also deploy dynamic taint-tracking to aid analysis, however, we instead taint state memory, rather than input messages. More significantly, as state machines can only incorporate as much behaviour as their training data (or queries), Prospex's passive learning approach is likely to miss 'interesting' protocol flows that for example trigger erroneous behaviour. On the other hand, our active approach achieves much greater coverage, which gives a more in-depth understanding of the implemented protocol, it's quirks, and possible flaws[1]. A further limitation of Prospex is that it fails to consider the state of run-time memory when classifying the effect that messages have on protocol state. This is crucial, as any states which differ only by e. g. a counter that is incremented, would be indiscernible, even if paths via these states eventually let to more consequential behaviour (e.g. authentication bypasses or other vulnerabilities). We tackle this limitation head-on in Chapter 6.

Another notable grey-box PRE approach is that of MACE by Cho et al. [34]. The technique presented constitutes a hybrid of both language and grammar inference. In particular, state machine inference is used to guide symbolic execution in order to discover new input messages. These new inputs are then fed back to the process to help with model (or grammar) refinement. The process begins by learning a high-level state machine in the standard black box way. Then, in order to refine this model, concolic

[1]This does however come with the cost of building a test harness for the protocol, which is less appropriate when considering *unknown* protocols

execution is used to trace execution paths to each of the already-learned-states. This trace is then processed by the engine to record the branch conditions and the corresponding constraints enforced on the inputs. Once the sequence of inputs has moved the execution into a target state (i. e. a node of the Mealy machine model), one can backtrack up the constraint path in order to take a different path. The symbolic (SMT) solver can then be instructed to produce inputs which satisfy the negation of each constraint. These inputs are then tested against the current model hypothesis to see whether they help refine it. A key part of this algorithm is the filter function, which checks (against the target) whether the new inputs produce a sequence of outputs which has not yet been observed. If so, they are retained and used to refine the model, otherwise they are discarded. This way, redundant inputs (i. e. inputs part of the same equivalence class) are not considered. The benefit of this process is that by providing only a small number of initial possible inputs (which can for example be extracted from observed packet traces of the protocol) a basic model can be learned and then refined to an arbitrary granularity through the iterative symbolic execution procedure.

Although an innovative approach for model learning and PRE, the fundamental components of MACE present difficulties in the application to security protocols. Firstly, MACE is not capable of learning protocols which utilise cryptography. This is due to its reliance on symbolic execution, which cannot function when provided with constraints produced by cryptographic functions. Clearly, this alone means the technique is not appropriate for the security protocols analysed by this thesis. This aside however, it is unclear whether the aforementioned filter function proposed by MACE, is effective for uncovering logic flaws in protocols. This is because each newly discovered input is filtered out if it does not produce a unique output sequence. The implication of this is that even if an input which could by used to e. g. bypass an authentication step (which by definition would produce a non-unique output sequence), were discovered, it would be discarded. Despite these limitations, we believe integration of MACE with the StateInspector work of Chapter 6 may help in overcoming its restricted applicability. We discuss this further in the future work section of Chapter 7.

On the other end of the spectrum, Chen et al. [33] recently proposed an ambitious approach to state machine extraction through the use of static source code analysis. While a static approach is useful for reasoning about all possible execution paths, it cannot provide precise information about input/output behaviour, or dynamic properties such as that of heap-allocated memory. Additionally, strong assumptions are made about the location of state transitioning logic, meaning that state changes which take place in recursive functions, event handlers or indirect calls cannot be detected. These limitations combined make the technique more appropriate for extracting basic string-like parsing state machines.

3.3.3 Fuzzing

Aside from analysing protocols in order to understand *how* they operate, many past works have considered the question of whether they are *secure*. Although this too is the goal of the model learning work in this thesis, *fuzzing* tends to focus on a different class of security issues. That is, memory corruption style vulnerabilities such as buffer overflows, rather than flaws in the implemented logic.

With respect to the broader domain of fuzzing, coverage-based grey-box fuzzing tools like AFL [126] have proved to be dramatically successful over the years against a wide variety of targets. Applied to protocol implementations however, these tools are less effective, as fuzzing test cases do not typically factor in state. In order to explore deep into the state space of a protocol, knowledge of message structure and ordering is essential. To this end, tools such as Snooze [13] and Peach Fuzzer [48] (black-box) and Sulley [10] and BooFuzz [92] (white-box) operate by providing a means to manually specify message structures and orderings. Chen et al. [32] go one step further in their white-box approach and instrument AFL with (manually-identified) program locations where state changes occur. A novel fuzz-case prioritisation strategy is then deployed to guide AFL between different states based on fuzz yield feedback. This facilitates inter-state fuzzing without the need for pre-defined message orderings. Although in our work of Chapter 6 we are not concerned with fuzzing, our technique, in contrast to that of Chen et al., does allow for automatic identification of state memory and where writes to it occur (i. e. state changes). A possible combination of the two approaches therefore may warrant further investigation.

Black-box fuzzers which do not rely on the presence of protocol specifications nor source code have also been proposed. One such example is Pulsar by Gascon et al. [56]. To infer models of unknown protocols, Pulsar builds on the probabilistic techniques of PRISMA [74]. Pulsar gathers recorded network traces of the target protocol and machine learning clustering techniques (c. f. Prospex's discussed grey-box approach [40]) are then used on both message structures and message ordering, resulting in a Markov model approximation of the implemented state machine. Minimisation to this model is applied to produce a more faithful representation of protocol logic in the form of a Deterministic Finite Automata, which can then be used to orchestrate fuzzing campaigns. Like other previously mentioned works, Pulsar's success is heavily reliant on the the quality of recorded traces. This means it cannot be considered suitable for stress-testing the logic of protocols like in this thesis. Moreover, as it relies on the ability to replay messages from traces during fuzzing, the approach unfortunately does not work in the presence of encryption.

Other recent work has also looked combining fuzzing with the active model learning algorithms discussed in Section 3.2 (e. g. the L* or TTT algorithms used in this thesis). In the Masters thesis of Janssen [70] from 2016, this combination of techniques is deployed in order to aid understanding of programs that are heavily obfuscated. In the equivalence checking phase of learning (ref. Section 2.2), counterexamples are identified through fuzzing and fed back to the learner to refine the state machine. This idea has been further developed in the recent AFLnet tool by Pham et al. from 2020 [95]. This is a promising development which, similarly to the aforementioned work on MACE [34], tackles the limiting exploration performed with fixed input sets like in this thesis. However, further evaluative work is required on this topic in order to establish applicability to complex protocols like TLS and WPA/2. Moreover, like MACE, the same limitations apply with respect to modelling states solely with I/O information—something we highlight in Chapter 6. We discuss this work further in future work (Section 7.1).

Part II

Black-Box Implementation Analysis

Chapter Four

Verifying Implementations of TLS Server Authentication in Mobile Apps

In this chapter, a novel approach for detecting flawed certificate pinning implementations is presented. The technique is a targeted variation of the central theme of this thesis, which looks to infer protocol behaviour from the interactions it has with its environment—in this case, the network I/O.

4.1 Motivation

TLS is a tricky protocol to get right: both misconfiguration vulnerabilities (e.g. [49, 86]) and attacks on the protocol are common (e.g. [17, 44, 87, 19]). In recent years, especially in mobile applications, a security measure known as Certificate Pinning, has seen increased adoption. The technique is designed to mitigate risk of not only trust-store compromise (which are often unnecessarily bloated [93] in size), but also compromise of individual CAs. The core idea behind Certificate Pinning is for developers to choose to only accept TLS certificates signed by a single *pinned* CA root certificate. Alternatively, but at the cost of reduced flexibility, a leaf certificate can be pinned. Although developers have been aware of the technique for some time now, in 2015 Oltrogge et al. [90] found general developer understanding to be poor due to the complex implementations available. Despite this, Chothia et al. [37] discovered that a large proportion of the high security UK banking apps they tested were implementing pinning. Focusing their analysis on these apps, they considered the possibility that apps which pinned to a CA root certificate, correctly checked that server certificates were signed by this root CA, but failed to verify the hostname.

A number of approaches for identifying flawed certificate verification do exist. Static code analysis techniques are appealing because in theory they can be fully automated. In practice however, best efforts have only succeeded in identifying "non-standard" implementations, for which manual dynamic analysis is then performed as a follow up [49, 111]. Typically, dynamic approaches include: ① serving self-signed certificates to clients to identify lack of signature verification, ② adding custom CA certificates to the device's trust store can detect missing certificate pinning, and ③ serving self-obtained (through purchase or otherwise) leaf-certificates signed by pinned CAs to detect pinning with missing hostname validation (as in [37]).

We motivate the need for an alternative approach to part 3 of the above, by considering the practicality of procuring the required certificates in order to carry large-scale testing of flawed pinning implementations. In particular, one would need to purchase a certificate from every possible CA for their domain. If we consider only a single CA—Symantec, who use 14 intermediary certificates to sign domain leafs[1], it would cost approximately \$17,000 per year where each certificate costs \$1,200 on average. Scaling to incorporate all possible other CAs used by app domains clearly incurs a high, prohibitive cost. To make matters worse, the process of obtaining some of these certificates is far from straight forward. High security apps will often use Extended Validation (EV) certificates. EV guidelines state that these certificates must only be distributed to registered

[1]As of the time of writing in 2017.

businesses and mandate a number of identity and legal checks. Though fulfilling these requirements for e. g. a pentesting company, is possible, certificate authorities may be reluctant to issue multiple EV certificates to the same organisation.

4.2 Contribution

In this chapter we present a black-box method to detect apps (or devices in general) that, when using TLS, pin to a root or intermediate certificate but do not check the hostname of the host they connect to. Instead of trying to get certificates from all possible certificate authorities, which we argue is infeasible, we build a tool that makes use of the Censys Internet scanning search engine. Given the certificate for a target domain, the tool queries for certificate chains for alternate hosts that only differ in the leaf certificate. The tool then redirects the traffic from the app under test to a website which has a certificate signed by the same CA certificate, but of course a different hostname (Common Name). If the connection fails during the establishment phase, then we know the app detected the wrong hostname. Whereas, if the connection is established and encrypted application data is transferred by the client before the connection fails then we know the app has accepted the hostname and is vulnerable. The key insight here is that although we cannot decrypt the traffic, the information we need is provided by analysing exactly when and how the TLS communication fails. While the tool itself is fully automated, we refer to the testing framework as semi-automated, as the user still needs to install and run the app by hand (this is something that could also be automated using an app emulator).

Using this tool, we carry out a test of 400 iOS and Android high security applications including banking, stock trading, cryptocurrency and VPN apps. We find 9 new apps that pin to root or intermediate CA certificates but fail to verify the hostname, rendering them all vulnerable to Man-in-the-Middle attacks. We note that the total user base of these apps is tens of millions of users, which highlights the severity of the issue.

We then reverse engineered each of the vulnerable apps in order to get a better understanding on how this weakness is introduced by the app developers and how to prevent it.

4.3 A Framework For Automatically Detecting Missing Hostname Verification

The key elements of our framework are: (1) a method for looking up the domain of websites that use a given certificate chain, (2) a custom built DNS server that will let us whitelist some URLs and redirect others to an IP of our choice, and (3) a TLS proxy that will redirect TLS messages and inspect the handshake to determine whether clients accept of reject the connection.

To find domains running TLS servers with specified certificate chains, we make use of the Censys[2] search engine, which hosts information on internet hosts by carrying out daily scans of the IPv4 address space [46]. By analysing the certificate chain in use by a domain that the app attempts to communicate with, the tool extracts the Common Name value embedded in the issuing certificate (either an intermediate or root certificate). This value is then used to construct a *Certificate* API query to Censys, which when executed, provides the tool with a list of domains corresponding to our requirements. The tool then selects the first reachable domain and communicates this to the TLS server accordingly.

As an alternative for when Censys is not accessible, we additionally provide a static database containing a mapping of issuer certificates to a set of hostnames in possession of a corresponding certificate. This was built by scanning the certificate chains in use by the Alexa top 1 million websites. In total, our database contains 33,601 unique issuing certificates, which easily satisfied the requirement for all the apps we tested.

[2]https://censys.io/

The DNS component of the tool is used to redirect the messages from the apps being tested to the TLS proxy. This proxy will in turn then redirect the apps traffic to a site selected from either Censys or the database. The TLS handshake is then analysed to look for indications that the client successfully connects to the un-requested TLS server.

In Figure 4.1, we detail variations of how TLS handshakes can fail. Consideration of these variations is important such that our tool accurately classifies TLS connection success, and thereby whether certificate verification was carried out correctly. Six separate scenarios (a, b, c, d, with two variants of c & d) were identified from experiments on the 400 apps in our test set. Variants 4.1.a) and 4.1.b), depict behaviour from apps which immediately terminate the connection upon receiving the unexpected certificate. As shown this is done either by sending an explicit TLS Alert message, or by just silently ceasing communication. The remaining scenarios depict implementations which seemingly defer certificate verification until later on in the handshake, specifically once the two parties had switched to encrypted message exchange (i. e. after the ChangeCipherSpec messages had been exchanged).

Interestingly, in cases c) and d), we further observed a slightly different breakdown in the connection. That is, in the time just before a client sent an alert message and then closed the connection, the server would send some encrypted application data. This bears the important consequence that the framework must only detect a successful connection if a client sends encrypted application data.

Using these experimental findings of how client implementations react to unexpected certificates, we configured our framework to flag successful connections through observations of encrypted application data from the client to the server, and anything else as indicative of a failed connection. Using all components described above, our framework proceeds to test apps using the follow process, split up into the proceeding three stages below.

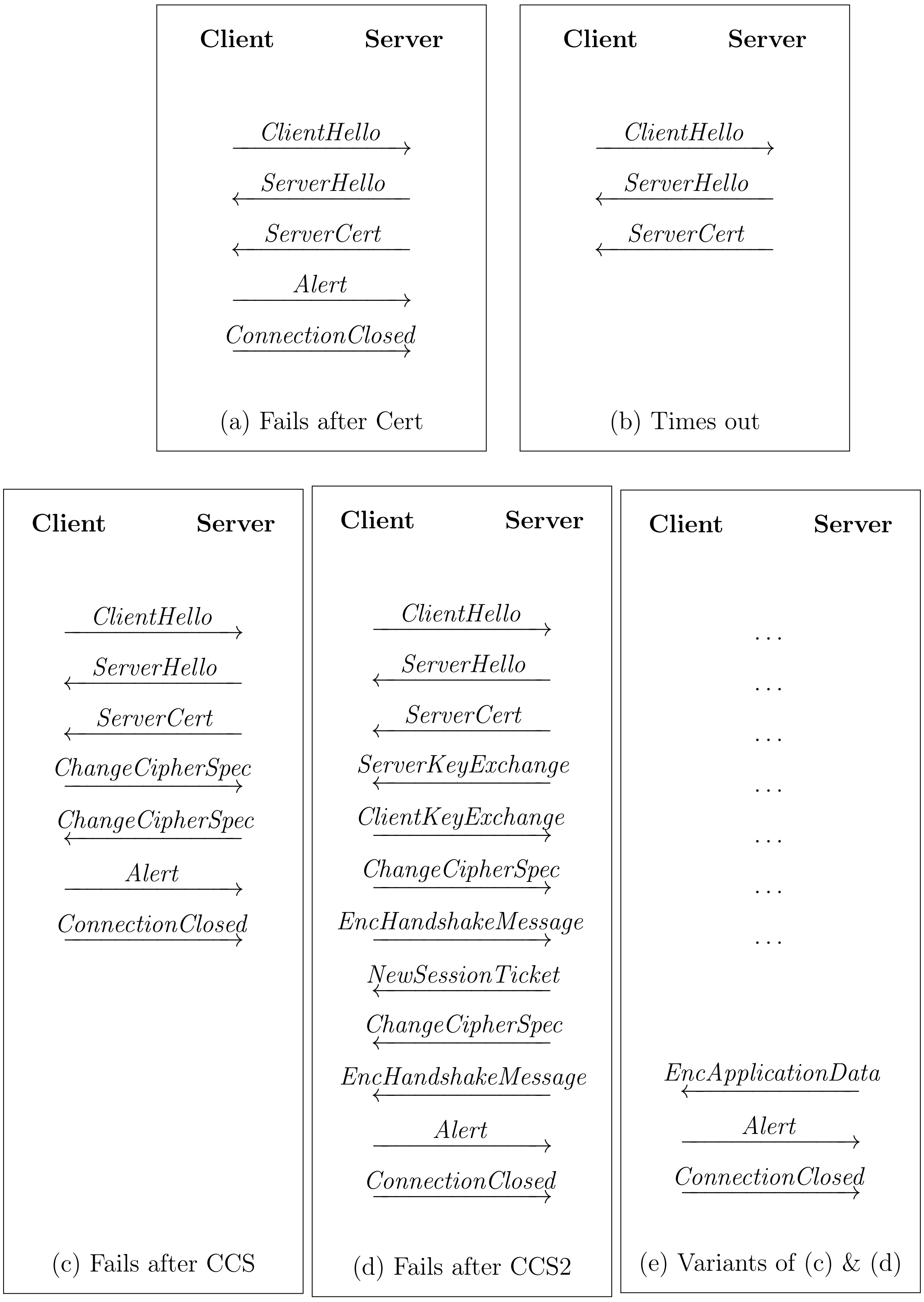


Figure 4.1: Observed failure scenarios of the TLS handshake

DNS Spoofing

1. We begin by installing the target application on the phone and then launching both the DNS server and TLS proxy elements of the framework. We then configure the phone to use the IP of our framework as its DNS server.[3]

2. We then manually launch the app and perform UI interactions in order to trigger attempts to initiate a TLS connection.

3. Prior to any TLS connections, the app will perform any required DNS lookups to our framework.

4. Given the requested domain, our framework will use a legitimate DNS server to look up the real IP address. The framework will then connect to this IP address and obtain the servers TLS certificate. Using this, it then queries Censys (or the static database) to find the URL for another site which uses the same certificate chain as the site the app wished to talk to. We note that some apps query for IP addresses for multiple domains at once, before any connections are made. Therefore, a mapping of requested hostnames to redirect domains is maintained.

5. The framework then responds to the app's original DNS request with the framework's own IP address.

TLS Proxy

Once the app knows to direct its traffic to our tool's IP address, the following process is executed. Each connection to each domain is tested in isolation. To achieve this, all other connections are whitelisted and proxied to their legitimate servers.

[3]An alternative is to configure the HTTP proxy settings of the phone, however apps often bypass this, whereas they do not for DNS.

1. Our TLS proxy inspects the SNI contained in the Client Hello message of incoming TLS connections. This value is used to distinguish connections, and is also used as a lookup to chose which domain to redirect to.

2. The Client Hello and all subsequent TLS handshake messages are forwarded to and from the server it looked up (which will be using the same certificate chain, but with a different hostname).

3. The proxy then examines how the app reacts to the certificate with a different hostname. If an alert is observed as in 4.1.a) or a time out occurs, as in 4.1.b) then we know the app successfully performed hostname verification.

4. If on the other hand we observe the app sending encrypted application data, then we know the handshake succeeded and the app failed to verify the hostname.

5. If no more connections to test, finish. Otherwise, white list the current domain being tested, and repeat test with next domain.

Vulnerability Identification

For each connection that the app makes, which is found to succeed when proxied to the alternate domain, the following process is carried out to pinpoint the exact vulnerability.

1. To establish whether the app accepts self-signed certificates and does not check the hostname (i. e. does not check the server's certificate is signed by a trusted CA), we redirect the TLS connection to https://self-signed.badssl.com[4]. The test fails if our tool detects a successful TLS connection.

2. If the previous test fails, we know that the app either does not check the hostname and uses the devices trust store for signature verification, or, the app pins to a particular certificate in the trust chain but also fails to verify the hostname. To

[4]BadSSL.com is a site hosted by Google for basic testing of clients against incorrect TLS config.

identify whether the former is the case, we redirect the traffic to a domain that uses an entirely different certificate chain.

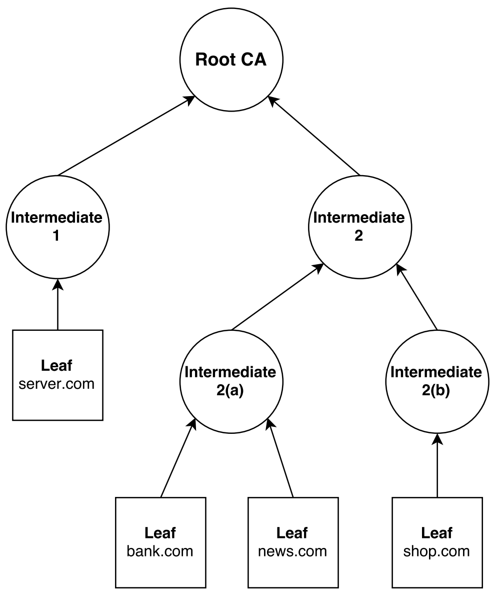


Figure 4.2: Typical hierarchical PKI, with four possible certificate chains

3. If the app is found to pin and not verify the hostname, the tool executes additional tests to determine exactly which certificate in the chain is being pinned. This done by iteratively querying Censys for domains that have certificate chains that vary in intermediates, but share the same root. For example, consider Figure 4.2, which illustrates four different certificate chains originating from the same root CA. When testing an app that connects to bank.com, the first round of tests proxies the traffic to news.com, which we find is accepted. According to Figure 4.2, traffic would then be proxied to server.com to establish if the Root CA is pinned. If the test passes, we know the app must pin to either Intermediate 2 or Intermediate 2(a). To narrow down further, traffic would then be proxied to shop.com. If successful, then the app pins to Intermediate 2, if not then to Intermediate 2(a).

App name	Platform	Certificate pinned	Certificate Type	Cost to exploit
TunnelBear VPN	Android	COMODO RSA Certification Authority	Intermediate	Free
Bank of America Health	Android	VeriSign Class 3 Public Primary Certification Authority - G5	Root	\$366
Meezan Bank	Android	VeriSign Class 3 Public Primary Certification Authority - G5	Root	\$366
Smile bank	Android	AddTrust External CA root	Root	Free
HSBC	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	\$995
HSBC Business	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	\$995
HSBC Identity	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	\$995
HSBCnet	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	\$995
HSBC Private	iOS	Symantec Class 3 EV SSL CA - G3	Intermediate	\$995

Figure 4.3: Apps in our test set that pinned but lacked hostname verification. We specify the particular certificate which is pinned in each app, and whether this certificate is a root or intermediate. We also denote the cost to purchase a a new certificate signed by the respective pinned certificates, according to market prices in 2017 (a necessary step to exploit the vulnerable apps).

4.4 Results

To try out our framework, we tested 250 Android and 150 iPhone applications. We choose to look at categories of apps that are often considered to be high security. This included banking, trading, cryptocurrency and VPN apps. We selected these categories as they constitute security critical apps which are likely less susceptible to simple attacks (e. g. accepting self-signed certificates). Furthermore, many of these type of apps will pin their certificates (a precondition for this particular attack).

For each app category we considered, the top 20 apps by download numbers were installed and tested. The remaining apps were chosen by browsing through the appropriate categories in each of the app stores (Google Play and iOS). We downloaded apps from the US, UK, Belgium and Indian app stores. However, a significant portion of these are apps that are available globally.

We ran our framework, and manually opened each app in turn. Of the 400 test apps, we found 24 which sent encrypted application data to the redirect-domain selected from Censys. From this set, 6 were identified as missing basic certificate validation as they accepted self-signed certificates; these were all banking apps from developing countries, some of which have been reported before [101].

Our framework then ran further tests on the remaining apps to discern whether they were implementing certificate pinning. This process is described in Section 3(b) of the vulnerability identification stage. Of the 18 apps that only accepted certificates signed by a trusted CA, 9 were found to not pin and hence we deduced they would accept any valid certificate for any hostname.

The remaining 9 apps were all found to be pinning the intermediate or root certificate but not checking the hostname. For each of these apps, our framework then went on to determine the exact certificate being pinned. A summary of the apps that failed

App name	Vulnerability	Platform
Emirates NBD	Self-signed	iOS
Kotak Bank	Self-signed	iOS
Al Rajhi Bank	Self-signed	iOS
Santander UK (biocatch)	No hostname check	iOS
CommBank Property	No hostname check	iOS
American Bank of Sydney	No hostname check	Android
Ulster Bank NI	No hostname check	Android
Ulster Bank RI	No hostname check	Android
BofAML Research Library	No hostname check	Android
First Financial Bank	No hostname check	Android
ACU Mobile	No hostname check	Android
Bitcoin.co.id	No hostname check	Android
Britline	Self-signed	Android
Opal Transfer	Self-signed	Android
Aman Bank	Self-signed	Android

Figure 4.4: Apps that additionally failed our tests.

our tests, along with the certificates they pinned are shown in Figure 4.3. If we regard the HSBC family of apps as a single code base, then the distribution of apps that pin to intermediate certificates in contrast to those that pin root certificates, is approximately the same.

For apps that are found to pin the certificate but not check the hostname, an attacker could then go to the trusted third party used by the certificate and obtain a valid certificate in their own name. This certificate can then be used by the attacker to trick a victim's app into thinking it is communicating with the server it expects, and hence decrypt and/or modify any sensitive traffic.

App name	Domains connected to	Vulnerable?
TunnelBear	api.tunnelbear.com	✓
	stream.tunnelbear.com	✓
	s3.amazonaws.com	✓
BofA Health	benefitsolutions .bankofamerica.com	✓
Meezan Bank	mbanking.meezankbank.com	✓
Smile	public-05.p01cd18.monitise.eu	✓
HSBC	services.mobile.hsbc.com	✓
	mapp.us.hsbc.com *	✗
	security.us.hsbc.com *	✗
HSBC Business	www.hsbcnet.com	✓
	www.business.hsbc.co.uk	✗
	www.secure.hsbcnet.com	✗
HSBCnet	www.hsbcnet.com	✓
	www.secure.hsbcnet.com	✗
HSBC Private	services.mobile.hsbc.com	✓
	www.us.hsbcprivatebank.com	✗
HSBC Identity	www.business.hsbc.co.uk	✓
	www.hkg1vl0077.p2g .netd2.hsbc.com.hk	✗

* This domain depends on the country that is selected when setting up the app.

Figure 4.5: Domains that are connected to.

Analysis

Past investigations of certificate verification flaws in mobile applications, such as Fahl et al [49], found that the acceptance of any certificate (including those that are self-signed) to be the most prevalent of this category of vulnerabilities. In 2012, Fahl et. al analysed 13,500 apps from the Google Play store. They found that 790 accepted any certificate, and a further 284 accepted any CA signed certificate for any hostname. In contrast, from our scan of 400 apps, we find that although the total proportion of apps that contain TLS flaws has reduced, the distribution of the types of flaws that vulnerable apps do have has changed significantly. We find that in fact lack of hostname verification to now be the most common flaw, and the variation of this that arises due the use of certificate pinning to be the second most common. Figure 4.4 shows all the other apps that were flagged as vulnerable by our tool, but were found not to pin to any certificate.

We took a random subset of 135 apps from our Android app data set. We then checked how many of these apps where using certificate pinning. This was done by adding our own root certificate to the phone's trust store, spoofing the DNS request and generating a certificate (on the fly) for the requested domain which is then used to MITM the connection. If the app accepts this then we conclude that it is not pinning.

Our experiments show that 38 out of those 135 apps implement certificate pinning, which is a much higher ratio than the results from Oltrogge et al. who found that 45 out of 639,283 apps used pinning in 2015 [90]. This increase could be attributed to a bias in our sample set, since it only includes high security apps; or it could be evidence of an increase in the use of pinning; or most likely, a combination of the two.

4.5 Vulnerability Impact

Having identified a number of apps that pin to intermediate or root certificates but miss the crucial step of hostname verification, our next steps were to evaluate the impact on security for each of these apps. An important consideration to be made is whether sensitive information, such as log in credentials, is sent over these vulnerable connections. It may be the case that connections to the vulnerable domains only contain non-sensitive data such as adverts or analytics (this can be avoided to some extend by white listing common analytics domains such as crashlytics.com and webtrends.com, however is obviously not possible for all domains). We therefore, adopted the following methodology to determine the seriousness of the vulnerability.

- Run our framework on each app, making efforts to trigger as much functionality of the app as possible through manual interaction. This gives us a list of domains the app connects to and whether these connections are vulnerable to TLS Man-in-the-Middle.
- For each app with at least one vulnerable domain, we judge whether connections that involve the transmission of sensitive data, are made to the vulnerable domain. This is done by checking whether the submission of random log-in credentials results in a connection attempt to that domain. If so, and the credentials are rejected, then we deem the app to be vulnerable to an attacker stealing these credentials.

Applying this methodology the all the apps that were detected by our tool (listed in Figure 4.3), we find that TunnelBear VPN vulnerable connections to three domains, whilst Bank of America Health, Meezan Bank and Smile all made single connections to the vulnerable domains when submitting log in or registration details. Each app and the corresponding domains it made connections to is listed in Figure 7.

Special Case: HSBC

The behaviour of HSBC's apps meant that its security impact analysis was more involved. On opening HSBC's main app, requests were made to the domain services.mobile.hsbc.com. We found that these connections were pinning to an intermediate certificate, but not checking the hostname. Despite this, as no interaction (other than opening the app) was made, we could not initially conclude that any sensitive data was being transmitted. Furthermore, when these connections were white-listed, and we were presented with the log in screen, a connection to mapp.us.hsbc.com was made. Additionally, any submitted credentials were seemingly sent to another domain security.us.hsbc.com. Connections to these two latter domains were found to be secure, i.e., passed our tool's tests. We did however find that they were not using certificate pinning, hence we were able to confirm the log-in details were sent to these non-vulnerable domains by installing a custom CA on the phone.

At this point one would deduce that users of the app are safe from having their credentials stolen by an attacker. However, it is possible that the app could be performing other sensitive operations, such as an update check. Moreover, the fact that pinning was being implemented solely for connections to this domain, suggested that in fact requests to this domain were of some significant importance.

We approached this problem by reverse engineering the iOS app. We searched for requests to the domain services.mobile.hsbc.com, and found that the app downloaded the resource /app/EntityList-1.5.17.xml. As the Android version was seemingly identical to the iOS version, including the same domains connected to (as well as pinning connections to the mobile.services.hsbc.com domain), we confirmed that this XML resource was also obtained, but named slightly differently: /app/EntityList-1.5.17-Android.xml.

Contained in this file are URLs for config files that we observed being downloaded after white listing connections to mobile.services.hsbc.com. We note that as the main HSBC app is available globally, the XML file contains different config URLs for each respective country where HSBC operates. A portion of this file is listed in Appendix A, Listing A.1. Inspecting the contents of these config files, we can see that the app uses this for a number of things including: update checks, with included download links and corresponding checksum values (HSBC uses hot code push/replacement for some of its updates); miscellaneous app content like in app text, images and adverts; links to web resources that open in the mobile browser; and, critically, the domain to communicate with during the log-in process.

As a result, we conclude that by exploiting the pinning vulnerability for connections to services.mobile.hsbc.com, an attacker can intercept and modify the downloaded config file to force a victim to send authentication details to a domain under the attackers control.

Attack Scenario

To carry out this attack, an adversary is required to man-in-the-middle the network connection of the victim. A example where this might be possible is when the attacker is on the same WiFi network, such as an airport or coffee shop. Using ARP or DNS spoofing, the victims traffic can be redirected to the attacker. Alternatively, a victim could be tricked into connecting to 'fake' hotspot set up an attacker in what is known as an "evil twin" [14] attack. When the victim attempts to use their vulnerable app, the attacker can intercept the TLS handshake and provide the app with a certificate signed by the certificate that the app pins to. Figure 4.3 shows the cost attached to carrying out the attack for each app. Note that as hostname verification is not being done, only one of each certificate needs to be purchased. Therefore it would cost $995 to attack all of HSBCs vulnerable apps.

Disclosure

We have disclosed the vulnerabilities in the apps to all of the companies involved. The vulnerabilities in these apps have now all been fixed. We received varying degrees of responsiveness from the organisations we disclosed to. Some patched their apps within a few days, whilst others took longer. The fastest responses were from Smile bank and from CommBank fixing their property app. The longest patch cycle was from HSBC, as shown

22 May 2017	Vulnerability discovered
23 May 2017	Initial disclosure by e-mail to HSBC
25 May 2017	Conference call discussing details of vulnerability
6 June 2017	We notice that an update to the “HSBC Business” app has added the vulnerability, we e-mail HSBC
3 August 2017	We send a follow up e-mail to check on progress and we contact the UK National Cyber Security Centre (NCSC) who also contact HSBC
4 August 2017	We receive an e-mail from HSBC saying they are working on the issue
31 August 2017	Follow up conference call with HSBC and NCSC. HSBC confirms the issue and say they are testing a fix
14 September 2017	Patch rolled out for main HSBC app

Figure 8: HSBC Disclosure Timeline

in Figure 8. Fixing their app seemed to take longer due to the fact that the same app was available in 39 different countries' app stores, hence making testing the app much more complex. We would like to thank the UK's National Cyber Security Centre (NCSC) for helping us with the disclosure process.

4.6 Why Hostname Verification Fails

To find why the vulnerable apps were weak, and to find out if simple static analysis methods could have found these vulnerabilities, we reverse engineered the vulnerable Android apps. We did this using the JADX tool[5] that can unpack Android APK files and decompile Smali to Java source code.

TunnelBear On initial inspection, the TunnelBear app seemed very secure. It did not disable hostname verification, and went further, defining a custom built hostname verifier to be used by all connections. The code for this verifier is shown in Listing 4.1. This test is applied automatically by the API, for all connections, and it seems to verify that the host connected to is one of `s3.amazonaws.com`, `amazonaws.com`, `tunnelbear.com` or `captive.apple.com`. However, on closer inspection we found that this is not actually how the verifier method should work.

The verifier method is called by the API with a string giving the hostname the user has requested a connection with, and an active SSLSession object. The verifier should then check that the given string matches the name of the host that the SSLSession connects to. The TunnelBear code does not do this: instead it simply checks the value of the string given to it and ignores the SSLSession object. If this app tries to connect to "tunnelbear.com", and we redirect it to "evil.com", the verifier will be called with str equal to "tunnelbear.com" and an SSLSession object that connects to "evil.com". The

[5]https://github.com/skylot/jadx

```
public final boolean verify(String str, SSLSession
    sSLSession) {
    if (str.contains("s3.amazonaws.com")) {
        log("BearTrust", "Regular trust enabled");
        return true;
    } else if (str.contains("amazonaws.com") && !str.
        contains("s3.amazonaws.com")) {
        log("BearTrust", "API Gateway enabled");
        return true;
    } else if (str.contains("captive.apple.com") || str.
        contains("tunnelbear.com")
    || ("https://" + str + "/").equals("https://api.
        tunnelbear.com/")) {
        log("BearTrust", "BlueBear trust enabled");
        log("BearTrust", "BlueBear enabled, trying IP");
        return true;
    } else {
        logError("BearTrust", "Failed to verify hostname")
            ;
        return false;
    }
}
```

Listing 4.1: Reverse Engineered Code from the TunnelBear app

verifier will just check the string, find it does equal an expected string, ignore the session object and log the connection as trusted.

This represents an understandable confusion about how the API works. A quick manual inspection could easily miss this subtle error, and simple static analysis tools would also miss this: a verifier is present, and hostname checking is enabled. To tell that the hostname string and SSLSession object were not correctly compared would require complex information flow checking, and even without a direct comparison, code could still be secure, if it for instance checked the SSLSession object against a static string.

The Smile Banking App Inspecting the Smile banking app we initially find it does not specify a custom hostname verifier, instead using the standard `STRICT_HOSTNAME_VERIFIER` option (which requires that hostnames match exactly). Further analysis revealed however, that the code which handles the opening of an SSLSocket was indeed flawed. We show this code in Listing 4.2.

This code uses the `SSLSocketFactory` object from the Apache HttpClient Java Library to create a SSL connection. As Georgiev et al. [57] have previously pointed out, this object only checks the hostname when the protocol is set to be HTTPS or LDAP by, for instance, using the `HttpsURLConnection` object. In this code the hostname and port number are resolved into an Internet address on line 5, and the raw socket connects to this address. Therefore this code does not check the hostname, irrespective of any flags set.

The setting of the `STRICT_HOSTNAME_VERIFIER` flag in the Smile banking app suggests the authors of this app were aware of the issue of hostname verification, however they had misunderstood the subtleties of the API. After our disclosure the company fixed this error by explicitly adding a verifier object. This could also have been fixed by setting the protocol to HTTPS or checking the hostname manually. Because of the range of ways to use `SSLSocketFactory` safely, we currently do not know of any static checker that could detect these issues without a high false negative result.

Bank of America Health This app follows the same pattern as the Smile app: it uses the Apache HttpClient Java Library `SSLSocket Factory` with a raw internet address. It

```
    public Socket connectSocket(Socket socket, String hostname
        , int portNo,
    InetAddress inetAddress, int i2, HttpParams httpParams) {
        int connectionTimeout = HttpConnectionParams.
            getConnectionTimeout(httpParams);
        int soTimeout = HttpConnectionParams.getSoTimeout(
            httpParams);
        SocketAddress inetSocketAddress = new
            InetSocketAddress(hostname, portNo);
        socket = socket != null ? (SSLSocket) socket : (
            SSLSocket) createSocket();
        if (inetAddress != null) {
            if (i2 < 0) { i2 = 0; }
            socket.bind(new InetSocketAddress(inetAddress, i2)
                );
        }
        socket.connect(inetSocketAddress, connectionTimeout);
        socket.setSoTimeout(soTimeout);
        return socket;
    }
```

Listing 4.2: Reverse Engineered Code from the Smile Banking app

does define a hostname verifier object, which would have correctly verified the Bank of America hostname, however because the protocol of the connection is not set to HTTPS and the verifier is not explicitly invoked, this code never runs.

Meezan Bank On inspecting this code we found that it used the `HttpsURLConnection` object that would normally check the hostname. However, the `ALLOW_ALL_HOSTNAME_VERIFIER` flag had been explicitly set, which disables all hostname verification. This one vulnerability could have been found by static checking, because there is no safe way to use this flag, therefore its presence in the code indicates a weakness.

4.7 Discussion & Limitations

Reverse engineering these vulnerable apps has revealed a number of interesting and subtle misuses of varying APIs that are available for implementing certificate pinning. These errors contrast significantly with those discovered in previous studies. Apps failing to verify certificates in the study by Fahl et al. [49] were found to be doing so in obvious ways, such as disabling certificate verification all together. This indicated a fundamental misunderstanding of the importance of server authentication in TLS.

Our results demonstrate that this is no longer the case, the vast majority of apps no longer have these basic errors. However, the added complication of certificate pinning, which appears to be on the rise, has spawned a new class of vulnerabilities. These are more subtle and hence have not been detected by existing detection techniques. Moreover, our findings are testament to the feedback received from developer interviews carried out in [90], which found that although general understanding of pinning is good, implementation complexity has made it difficult to roll out.

Clearly, the abundance of pinning implementation options available to developers has played a role in causing these flaws to be made. Platform providers can make this less of an issue by providing standardised implementations with clear documentation. To this end, Google have introduced Network Security Configuration[6] in the Android 7.0 SDK. This provides a easy way to configure certificate verification, including the ability to specify certificate pins and associated hostnames in an XML file. If app developers make use of these standard implementations, instead of rolling out their own or using 3rd

[6]https://developer.android.com/training/articles/security-config.html

party libraries, these errors will be much less likely to occur.

One major limitation of our presented approach is it's *semi*-automated nature. Apps are tested through both manual installation and UI interaction in order to trigger TLS connection attempts. The consequence of this is that we were only able to test a limited number of apps. An alternative testing framework may seek to perform app emulation followed by a form of UI fuzzing and automatic input generation to trigger TLS connections and thereby fully automate the process. A major challenge of this however is generating intelligent input which passes structural validation checks (e. g. correct username format), which is more-often-than-not a necessity for instrumenting apps to make their most sensitive connections (e. g. user log-in).

Chapter Five

Extending Black-Box Protocol State Learning for Analysing WPA/2 Implementations

In this chapter, we take a wider look at how protocol behaviour can be inferred from input testing and observations of outputs on the network. Compared with the previous chapter, we adopt the more systematic approach of learning state machine models which capture implemented protocol logic. These models promise to provide a more complete way to assess security and adherence to standards, however we find off-the-shelf learning algorithms inadequate for protocols like WPA/2. We present an extension of these learning algorithms to handle protocols that are both prone to erroneous, non-deterministic behaviour and those that also define time related behaviour. We implement our proposals and present our findings from tests on a number of WPA/2 access points, including CVE-2018-0412 on a range of Cisco devices.

5.1 Motivation

Automated, systematic analysis of protocol implementations has proven to be an effective tool for security evaluation. Examples of this form of analysis include fuzz testing [13, 28], model-based testing [25, 119, 17] and protocol state fuzzing (also known as state machine inference or model learning) [1, 54, 44] (all of which are discussed at length in Chapter 3 of this thesis). Traditional fuzz testing looks to locate memory corruption vulnerabilities and is typically stateless. On the other hand, the latter two approaches focus on testing protocol logic and are necessarily stateful. Model-based testing, applied to TLS by Beurdouche et al. [17] and WPA/2 by Vanhoef et al. [119], seeks to test protocols through manual definition of their "ideal" model (as defined in the protocol specification), followed by careful generation of test cases that may trigger deviations from this ideal. This can be a long and arduous task, and requires extensive knowledge of the protocol specification to decide whether every possible test case should fail or pass. Model learning takes a different approach; aside from the input message set, nothing is initially assumed about the protocol. Instead, verification of protocol logic and security analysis is performed after a full state machine has been extracted. This process of learning the state machine is carried out in a fully-automatic, black-box fashion by systematically sending different sequences of messages and observing outputs. The resulting state machine takes the format of a Mealy machine (see Section 2.2, Chapter 2), which models states based on their responses to inputs. The benefit of this approach is that test case generation is fully automatic and complete. Furthermore, state machine learning automatically adapts future and successive test cases according to the results of previous ones. For example, if one particular message sequence discovers some erroneous state or unexpected output, it does not stop testing there. The algorithm will continue to explore the state space beyond this and therefore cover more ground than is possible with model-based testing.

In this work we utilise state machine inference in order to carry out a black-box analysis of implementations of the IEEE 802.11 4-Way Handshake protocol. This widely used protocol is the means by which authentication and session key establishment is carried out on IEEE 802.11 (WPA or WPA/2 certified Wi-Fi) networks. A naive application of learning to this protocol would however fail due to two key aspects. Firstly, protocols like WPA/2 define time-based behaviour, e. g. message retransmissions and timeouts, though, in general, time-based behaviour can be entirely arbitrary. In protocol settings, past studies had to artificially suppress time-based behaviour, as it can introduce both non-determinism (which is incompatible with standard state machine learning) and an explosion in learning complexity. Formal time learning algorithms have been proposed (see for example [59]) but are non-practical and have been omitted from model learning libraries, again due to their high complexity. Previous works tackled time in various ways, for instance, ignoring re-transmissions and manually setting timeouts for responses to ensure time behaviour is not triggered [52, 54], or mapping multiple outputs within manually specified times to single state transitions [44, 114]. The former technique disables time learning altogether. This is an appealing approach as the risk of non-determinism is reduced, however may result in not modelling important behaviour, which as we will see, could represent vulnerabilities. In the latter, timeouts are manually identified and multiple responses are merged into one, reducing the state space but again potentially missing important behaviour. Clearly then, the need to devise an automated approach to incorporate time based behaviour into models, without exploding learning complexity, is a desirable feature of model learning algorithms.

Secondly, the quality of the transmission medium and query interfaces can also affect the ability to learn a system. With protocols like WPA/2, a response might be missed and incorrectly marked as a timeout, or a query is not processed by the target and a retransmission occurs, which effectively makes the system non-deterministic. This is a serious problem for state-machine learning algorithms as they require the system under test to be completely deterministic, anything less and learning fails. Is is therefore clear that adapting these algorithms to handle inconsistent behaviour would extend the class of protocols for which model learning is applicable to.

5.2 Contribution

In this chapter we propose practical methods to efficiently learn protocol time behaviour and overcome non-determinism. To learn time behaviour we reduce the complexity by making reasonable assumptions about the operation of network protocols. We separate time learning into a secondary learning step. This enables us to first learn non-time based behaviour, without incurring the costly time-complexity that timeouts induce. Throughout this process, we run an error correction method that handles query-response inconsistencies, thereby ensuring learning termination. We implement these methods and use our tool to learn models of the 802.11 4-Way Handshake on 7 access points. Our results include the discovery of three vulnerabilities: two distinct downgrade attacks and leakage of multicast data. To summarise, our contributions are as follows:

- We adapt standard Mealy machine learning to infer common time based behaviour in protocols. This is done efficiently and without the need for complex timed automata modelling.

- We provide a practical method to overcome occasional non-deterministic behaviour in protocols.

- We implement our solution and carry out protocol state fuzzing of a range of 4-Way Handshake implementations.

Our tool, along with model diagrams and other information related to this work has been made available online [80].

5.3 Adapting State Machine Learning For Wi-Fi

5.3.1 Learning Protocols with Errors

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

Running our learning algorithm on 7 routers, non-determinism was reported for between 0.5% and 8% of queries (full details are in Section 5.4). This error rate means that most attempts to learn a router will fail before the state machine can be found. The errors were mainly due to either a message not being received and the response timing out, or a message not being received and a previous message being retransmitted. In the latter case, there is no way to tell from a single response alone if the message is a genuine reply to a query or if it is a retransmission due to a lost message.

To handle this apparent non-determinism we maintain a record (or cache), which records all input sequences, all corresponding responses, and the number of times those inputs and responses have been seen. This component of the algorithm is made completely independent from the "backend" learning framework. This means that error correction support can be plugged into any learning algorithm or library. The only requirement we have is that the backend signals when non-determinism has occured. In practice, we use the popular library `LearnLib` [99] which does indeed throw an exception when a series of inputs gives a different output to one we have previously seen. We can then handle this exception, and execute a form of 'majority vote' error correction in order to decide on the correct response. This works as follows:

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

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

Applied to our WPA/2 case study, on average, we require in the region of 1000 queries to learn a model. Our method, and optimisations, leads to queries being executed an average of 15 times[1]. Assuming the highest error rate we saw of 8%, this means that the chance that we learn an error response, rather than the correct response for any query is less than 0.01% (see Appendix B). Working backwards from the failure probability,

[1]This is because queries iteratively increase in length, so the same prefixes for longer queries will be executed many times, with the number inversely proportional to the length of the query.

we find that our method will have a 95% confidence of returning the correct automata for error rates of up to 10%. Higher confidence and higher acceptable error rates can be achieved by retrying queries that are not strictly needed by our method, e.g., if we repeat queries to ensure that they are tried at least 100 times we can provide 95% confidence of learning an automata correctly for error rates of up to 30%.

When an error response becomes the most common response to a particular query, our method will discard useful information and thus be inefficient. For the worst error rate we observed, 8% we calculate the probability of discarding a correct response to a query with 15 tests as 0.756%, more tests do not increase this probability significantly. On the other hand an error rate of 30% would lead to a 18% probability of discarding useful data. We note that for such high error rates we could cache the learn queries rather than discarding them to avoid having to relearn responses.

5.3.2 Learning Time-based Behaviour

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

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

As previously mentioned in Section 3.2.1, Verwer et al. [120] provably determined the learning of multi-timer systems to be infeasible. As such, we limit the number of timers so that there is never more than one timer running at the same time. Indeed, for the purpose of learning the 802.11 handshake, support for single clocks was sufficient.

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

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

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

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

We arrived at these three assumptions both through our initial experiments of black-box learning on various access points, and code inspection of the open-source Linux based `hostapd`[2]. All our black-box tests indicated that any time a frame was re-transmitted, no observable change to state was made (with respect to responses to inputs). In `hostapd`, retransmission behaviour is triggered upon the expiration of a timer. This timer is configurable but typically lasts a second, and defines a callback to be executed named `wpa_send_eapol_timeout`[3]. Each time a (EAPOL) message is sent by the function `wpa_send_eapol`[4], this timer is registered, and if the expected response is not received before expiration, the callback is executed. Here, a timeout flag is set, and the

[2]https://w1.fi/hostapd/

[3]`src/ap/wpa_auth.c` line 1415

[4]`src/ap/wpa_auth.c` line 1658

main state stepping function that controls state transitions, `SM_STEP`[5], is called. In this function we can see that two key pieces of code are executed. Firstly, a timeout counter is incremented and checked against the configured max timeout count value, ending the handshake if this is exceeded. Secondly, if the aforementioned timeout flag has been set, the same logic executed to enter the *current* state is re-executed, thereby triggering a retransmission[6]. This in particular supports Assumption 3. We further note that although `hostapd` supports the registering of multiple timers, the handshake logic is implemented such that at no point these timers run simultaneously.

In addition to the above assumptions, we also assume that the modeller is able to provide estimated values for a normal response time and upper-bound timeout. The normal response time should be large enough to give the SUL sufficient time to provide non-timer based responses. Essentially, as long as it takes the SUL to receive and process a message, and send a response. In Wi-Fi, we set this in the region of 200–500ms. For other protocols, or testing set ups, the value should be set according to the quality of the medium on which the protocol is running. For example, one could conceive of a protocol running across further distances, and as such require a longer time allowance for single input/output queries. The second value is an upper-bound timeout. This is required to prevent the learner waiting endlessly if there has been a silent timeout. It should be sensibly set to a maximum value that you expect the SUL to maintain a connection for. E.g., we set this to 20s, as we expect any timers to have expired and connections to be dropped if the handshake has not completed within that time.

Solution Overview

In our solution, we split the learning procedure into two stages. The first stage will discover behaviour such as the normal flow of the handshake, and states unrelated to time. That

[5]`src/ap/wpa_auth.c` line 1415

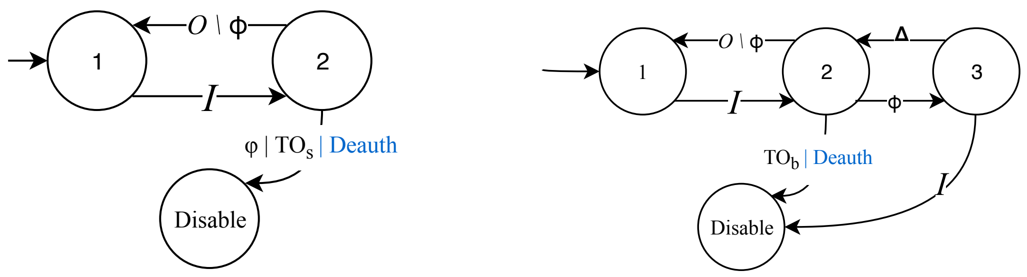
[6]The only difference in the retransmitted frame is an incremented Replay Counter

is, we first build a *base* model, which we can later use to learn time behaviour. This way, we can carry out extensive and thorough equivalence checking of the base model, without triggering long timeouts—which causes a blow up in learning time-complexity. The latter stage then uses the base model to identify time-based states, including retransmissions, timeouts and anything else. To this end, we employ two measures. First we use the cache described in Section 5.3.1, which records all query/responses in a database. This enables us to separate each stage of the learning. Second, we adopt the I/O automata learning method presented in [7]. That is, we employ a transducer that translates the non-Mealy-machine compatible SUL behaviour into sequences of query-responses that the Mealy machine can understand. This technique enables us to enforce learning restrictions for each corresponding stage. The transducer is implemented as a state machine itself, namely a Learning Purpose (LP). We construct the learning purpose such that it enforces the following restrictions on the types of permitted queries.

1. Each input symbol $i \in I$ constituting a query maps to a single output from the set $O \cup \{TO_s, TO_b\}$. Here, TO represents a timeout, which is set to the normal response time for the first stage (TO_s), and to the upper-bound timeout for the second (TO_b).
2. If a retransmission[7] is observed, we disable all inputs. An exception is made in the second learning stage where we allow the delay action Δ beyond this point.

The learning purpose representing the described properties for each corresponding stage is depicted in Figure 5.1. We can see that the learner will begin at state 1, where any input is enabled. From there, the resulting output O from the SUL will determine the next transition and so on. As soon as the *disable* state is transitioned to, any subsequent inputs will be disabled, meaning that corresponding outputs will be the '$-$' symbol. We include an optimisation of this feature which is detailed in Section 5.3.4.

[7]Retransmissions definitions can be customised. For the purpose of testing Wi-Fi, we define a retrans-



(a) Stage one

(b) Stage two

Figure 5.1: Learning purposes used for each learning stage. The two timeouts are indicated by TO_s, TO_b, ϕ indicates the last accepted response (retransmission), and Δ a delay action.

Compared to the previously mentioned approaches that merge multiple outputs to single transitions (within a fixed time), our approach represents a formal method of incorporating output sequences consisting of multiple messages, including re-transmissions, into the state machine model. This is done such that states are individually represented (i.e. each transition only retains the property of indicating a single output).

When testing 802.11 handshake implementations, we can make some adjustments to the learning purpose to improve efficiency further. Since we know that a `Deauthentication` indicates a reset of the protocol, we can disable any queries which trigger this output. This modification is highlighted in blue in Figure 5.1.

Stage 1 Learning We begin by running learning with the full alphabet and the learning purpose from Figure 5.1a enforced. Once a hypothesis has been produced, we run Chow's W-Method [38] for equivalence checking. This guarantees all states within the restriction of the learning purpose will be discovered (given an upper bound on the number of states). On average this stage will complete quickly, as all time based behaviour (which dramatically increases the execution time of each query) is ignored. Any counter-examples discovered in this stage will be recorded in order to reconstruct the model in

mission to be an identical message as before, with the exception of the Replay Counter value.

the second stage.

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

1. Firstly, we delete all entries in the cache oracle that have resulted in the small timeout TO_s. When learning is restarted, these deleted entries will be posed to the SUL again, this time with the new learning purpose from Figure 5.1b.
2. Learning is restarted using the new learning purpose. Each query in this stage will first check the cache oracle to see if there is a corresponding response from the first stage. Once a hypothesis has been conjectured, we apply the same counterexamples learned in stage 1.
3. We then begin an equivalence checking stage with the intention of learning all timeouts. That is, for each state already learned, we simply pose queries consisting of many delay actions. The resulting model will represent the base model from stage one, with time based behaviour included. Any non-retransmission, timeout or disconnect states disovered in this stage will also undergo further equivalence checking.

5.3.3 Broadcast/Multicast Traffic

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

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

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

5.3.4 Additional optimisation

Query Disabling The constraints that we enforce with the learning purpose (see Figure 5.1), such as disabling any queries beyond a deauthentication or timeout, can be exploited for further efficiency gain. Namely, if we ever observe a query response containing an disable output ($-$), then we know that any additional inputs beyond that point will also have the 'disable output'. This enables us to maintain a cache of all queries and their corresponding responses that result in the learning purpose transitioning to the disable state. This cache can then be used as a lookahead oracle for further queries. For example, say the query $q = \{assoc, delay, data, data\}$ results in response $r = \{accept, E1, timeout, -\}$. The lookahead oracle can then record this query-response pair, as it ended up in a disable state (indicated by the fourth output $-$). If then, the learner poses the query $q_2 = \{assoc, delay, data, data, E4\}$, we already know what the response will be because q is a prefix of q_2, and q ended up in the disable state. Therefore, we can automatically generate the response $r_2 = \{accept, E1, timeout, -, -\}$ without actually querying the SUL.

WPA/2 Specific optimisation In Section 5.3.2, we show how exploiting our prior knowledge of observing the `Deauthentication` frame, to indicate a reset of the protocol, can be used to improve learning efficiency. Similarly, we also implement a check which disables queries after a successful handshake/connection has completed and verified.

5.3.5 4-Way Handshake Input/Output Learning Alphabet

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

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

Where `MsgType` is one of {`Association,EAPOL2,EAPOL4`} and has associated parameters defined in the table below. Associations only permit the RSNE parameter, whereas for EAPOL-Key messages, it can be any. We also include the Delay action (Δ), (Unencrypted) Data, and Encrypted Data (TKIP and AES). We denote the Broadcast/Multicast Delay input (described in Section 5.3.3) as BRD in our models. In total our input alphabet consists of 45 unique messages. We note that a complete set of all possible combinations of the various EAPOL fields would consist of 1000s of frames. We therefore select the most important fields and values with respect to security.

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

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

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

Table 5.1: Parameter definitions for the 802.11 handshake input alphabet.

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

5.3.6 Implementation Details

Network Data When learning the state machines of our selected APs, we ensure that there is constant traffic, including unicast, broadcast and multicast, circulating on the network at all times. This enables us to learn broadcast and multicast traffic and also detect successful handshakes as mentioned below. We achieve this by operating a node on the network which run scripts to send: traffic directly to the fuzzer's MAC address (e.g. raw data), multicast traffic (e.g. using mDNS), and broadcast traffic (e.g. ARP).

Verifying and resetting connections As the last message of the 4-Way handshake is sent to the client and hence our learner, the corresponding response will normally be a timeout, therefore we need to distinguish between the case where a handshake has finished successfully and other kinds of time-outs. As mentioned in the previous section, we operate a node on the network that constantly sends unicast data addressed to our learner's MAC address. Therefore, once a handshake is complete we observe these messages and can decrypt them to verify the contents. If this succeeds, we then check that the fuzzer can itself send encrypted data. This is done by sending an ARP-request for the MAC address of the gateway IP and waiting for an appropriate response.

Multi-core/Interface Sniffing and Injecting Due to the unreliability of Wi-Fi monitor mode for 802.11 frame injection and sniffing, we use two physically independent interfaces for each task—sending queries and sniffing for responses. We then have two processes running in parallel so that sniffing and injection can be carried out simultaneously. This is all implemented in Python using the Scapy[8] library.

[8]http://www.secdev.org/projects/scapy/

5.3.7 Model Specific Analysis

In order to analyse the models learned by our tool, and identify any possible implementation specific vulnerabilities, we perform a manual inspection through comparison against an 'ideal' or 'expected' model. We have defined this model in Figure 5.2. In essence, this model captures our expectations that implementations should terminate the handshake if non-green (expected) messages are received at any state. Additionally, we expect there to be some time-behaviour like message retransmission (E1 and E3 at states 3 and 5 respectively) a limited number of times before the handshake eventually times-out. Behaviour outside of these expectations warrants further analysis as it could indicate a vulnerability. The results of our experiments and this analysis are presented in the next section.

We note that alternative, model-based verification approaches for analysing learned state machines have been proposed in other works (e. g. [52, 53]). These approaches work by defining a set of graph-based queries which check whether certain properties hold (e. g. there is only a single route from the start state to the authenticated state). Whilst these approaches are useful for large scale testing of many-state protocol implementations, we found that in our case a more direct manual analysis sufficed due to the compact nature of the resulting state machines.

5.4 Results

We used our adapted state machine learning algorithm to automatically learn 7 AP-side implementations of the 4-Way Handshake (see Table 5.2). In this section we will discuss the effect of our learning improvements, as well as the most notable results, including

[9]In the interest of brevity, we only include a selection of the most important transition labels in models presented in this thesis. Any queries that are 'disabled' are not included and we use the ∀ symbol to denote all input messages not specified in other transitions.

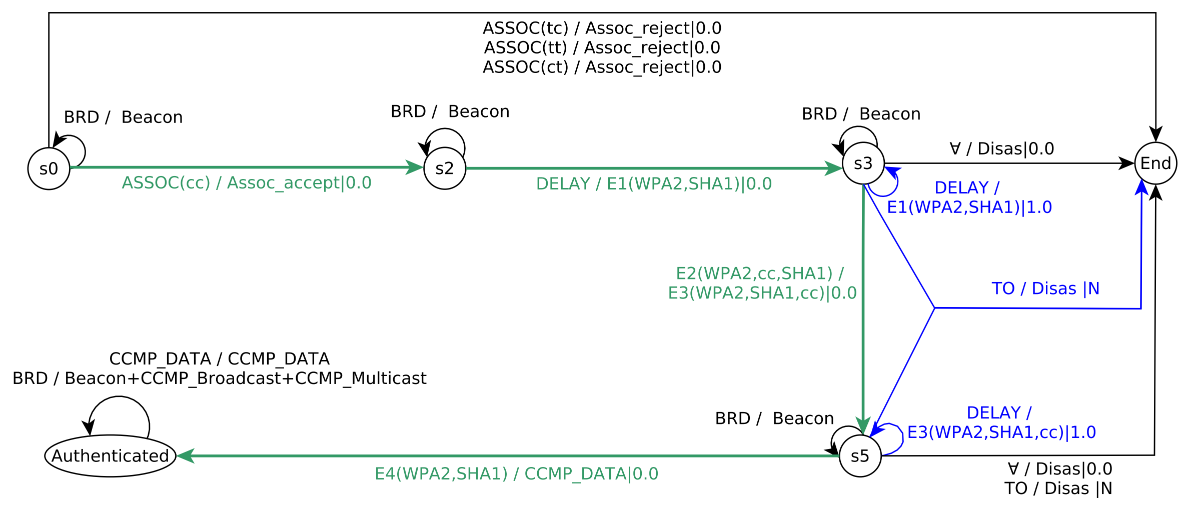

ASSOC(tc) / Assoc_reject|0.0
ASSOC(tt) / Assoc_reject|0.0
ASSOC(ct) / Assoc_reject|0.0
BRD / Beacon
s0
ASSOC(cc) / Assoc_accept|0.0
BRD / Beacon
s2
DELAY / E1(WPA2,SHA1)|0.0
BRD / Beacon
s3
∀ / Disas|0.0
End
DELAY / E1(WPA2,SHA1)|1.0
E2(WPA2,cc,SHA1) / E3(WPA2,SHA1,cc)|0.0
TO / Disas |N
CCMP_DATA / CCMP_DATA
BRD / Beacon+CCMP_Broadcast+CCMP_Multicast
Authenticated
BRD / Beacon
s5
DELAY / E3(WPA2,SHA1,cc)|1.0
E4(WPA2,SHA1) / CCMP_DATA|0.0
∀ / Disas|0.0
TO / Disas |N

Figure 5.2: A base model derived from expert knowledge and reference to specification documents. Green transitions indicate the normal, 'happy'-flow of the protocol. Blue transitions indicate expected time-based transitions, like retransmissions and timeouts.[9]

vulnerabilities, time behaviour and other interesting observations. We include figures of two of the learned models in this thesis, the rest are available online [80].

Time behaviour Three of the access points we tested exhibited the same timeout behaviour (3 retransmissions of message 1 and 3 with one second gaps, before ending with a deauthentication). Others had similar behaviour but over different times. One did not retransmit messages but timed out after 6s. Most interestingly, the Cisco WAP121 started sending encrypted data after 3 re-transmissions of message 3 over a period of 4s. We discuss this in more detail in Section 5.4.2. We note that this finding in particular could not be detected by previous methods. The iOS model stands out in that it took significantly longer to learn than the others. The reason for this is that it appears to silently timeout and hence hit the upper-bound timeouts mentioned in Section 5.3.2. Indeed, the implementation appears to be very minimalist, only responding to queries it considers to be correct, and ignoring those that are not. Nevertheless, this exemplifies the importance of the first stage of our learning method. By setting a small timeout (the 'normal response time'), when the learner carries out the equivalence checking stage, these

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

Table 5.2: Learning statistics for the Access Points we tested. Number of queries excludes those found as causing erroneous responses. Total learning time includes time taken for error correction. The error rate denotes the percentage of all queries which resulted in observed responses incompatible with the final model (i. e. immediately triggered non-determinism exceptions, or were corrected and discarded later on.)

queries will not suffer from this long timeout. Hence, thorough fuzzing was still possible, despite then having to relax this restriction for the second stage.

Non-determinism In Table 5.2 we state the error rate for each of the implementations we tested. We calculated this as the proportion of total executed queries that were detected as an error. An increased error rate had a direct effect on the time taken to learn. This is demonstrated by the TP-Link AC1200 which had an almost identical model to Android Hostapd, yet took over double the time to learn. In this particular case, the high error rate was due to the AP carrying over data from previous handshake executions with a relatively high probability. This suggested that resetting the AP between queries was not consistent.

Query reduction We were able to significantly reduce the number of queries required by the learning algorithm vs those actually posed to the SUL. Most of this reduction is down to the restrictions we enforce (i. e. delays after retransmissions (Section 5.3.2) and Wi-Fi specific optimisations (Section 5.3.4). For example, the iOS model required over 20,000 queries in total but only 887 were actually queried, the rest predicted.

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

5.4.1 Learning Without Time-Optimisations

To evaluate the effectiveness of our approach for learning time-behaviour, we ran the tool with only the error correction feature described in Section 5.3.1. Note that without error correction, learning was not possible for any of our selected APs as we observed a non-zero error rate for each of them. For a fair comparison, we took the approach of [44, 114], whereby we set a fixed timeout for responses to inputs. As done with these works, we set the optimum timeout on a per-implementation bases. For most implementations this was 2s. However, for the TP-Link WR841HP this was 6s, since we found from our previous tests that it timed out after 6s. Additionally for iOS, we set the same upper bound timeout as with our approach of 20s, due to the fact that timeouts are 'silent'. Note that our approach does not require assumptions about the specific implementation like this, but rather just a general maximum and minimum bound.

The Android hostapd implementation did successfully learn the same model as our approach, however this was done over the course of 49 hours and 35k queries. Similarly, this was the case with the Cisco WAP121 and TP-Link AC1200, which required slightly more queries due to the increased error rate. These cases demonstrate that for such targets, even though other approaches are possible, our approach may be preferable for analysts who require more immediate results.

For the D-Link and ZxYEL, learning was aborted after 3 days. With each of these, we found that error rates to increase over time, meaning many queries had to be tested multiple times. We conjecture that one of the reasons for this may have been resource exhaustion interfering with determinism. Occasional physical resets of the devices did improve the situation, however due to the long learning times involved, we decided stop learning. These particular findings show that our approach of optimised time learning, was able to mitigate this issue through a significant reduction in queries. We additionally aborted learning for the iOS Personal Hotspot implementation after 3 days. Although within that time the model produced was equivalent to the one learned with our approach, in this case the equivalence checking would not terminate within a reasonable time. This was because of the high average per-query time, since each input tested outside of the normal handshake flow reached the maximum 20s bound whilst waiting for a response. This meant that the longer equivalence queries took between 1m and 2m each.

Finally, learning the TP-Link WR841HP took comparable time in both methods. This was mainly because there was only one instance of time-behaviour—timeouts after 6s.

5.4.2 Vulnerabilities

Encrypted Multicast Traffic Leakage

Using the broadcast/multicast learning feature of our framework, we discovered that the TP-Link WR841HP transmits multicast data in plaintext when put in a certain state (see states 1 and 2 in Figure 5.3). More specifically, before a handshake is initiated, any multicast data will be sent encrypted with each unicast session key for all of the connected clients. However, during the execution of a 4-Way handshake with a new client, and before the client has proven knowledge of the PSK (by sending a valid Message 2), this data will broadcast unencrypted to the client. Indeed, immediately after the 4-Way handshake is completed, the data will only be sent encrypted. This represents a leakage of (potentially) sensitive multicast data.

Downgrade Vulnerabilities

In two access points we discovered downgrade attacks, namely for the Cisco WAP121 and the TP-Link TL-WR841HP. These attacks have long been an issue for security protocol implementations, with some of the first 'cipher rollback attacks' identified 25 years ago in early versions of SSL/TLS [121].

Forcing Group Key encryption with RC4 Figure 5.3 shows the learned state machine implemented by the TP-Link WR841HP. We can see that despite initiating the connection in the Association stage with AES-CCM for both group and unicast keys, after starting the 4-Way handshake using AES-CCM, the AP will surprisingly still accept a TKIP-formatted Message 2. In other words, if the client switches ciphers mid-handshake, the AP will do also. This is indicated by the AP's response from state 3 to state 5, where it switches cipher suites to use TKIP's MD5 for the MIC, and encrypting the network's group key with RC4. Of particular significance is that this is in spite of the AP being set to exclusively use AES-CCM. Indeed, this is also advertised in the AP's Beacon and Probe Response messages.

To exploit this vulnerability, the adversary can set up their own AP with the same SSID as the target. This AP, however, only advertises support for TKIP in the Beacon/Probe Response. As shown in Figure 5.4, the attacker simultaneously carries

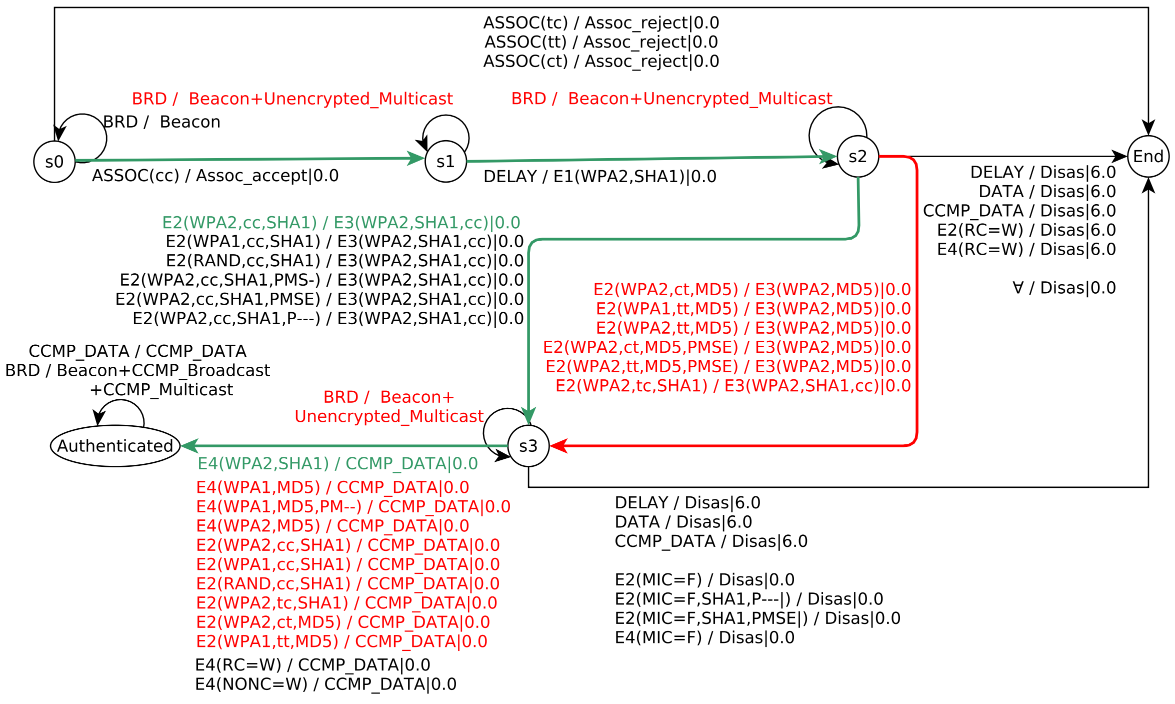

ASSOC(tc) / Assoc_reject|0.0
ASSOC(tt) / Assoc_reject|0.0
ASSOC(ct) / Assoc_reject|0.0
BRD / Beacon+Unencrypted_Multicast
BRD / Beacon
BRD / Beacon+Unencrypted_Multicast
s0
ASSOC(cc) / Assoc_accept|0.0
s1
DELAY / E1(WPA2,SHA1)|0.0
s2
End
DELAY / Disas|6.0
DATA / Disas|6.0
CCMP_DATA / Disas|6.0
E2(RC=W) / Disas|6.0
E4(RC=W) / Disas|6.0
∀ / Disas|0.0
E2(WPA2,cc,SHA1) / E3(WPA2,SHA1,cc)|0.0
E2(WPA1,cc,SHA1) / E3(WPA2,SHA1,cc)|0.0
E2(RAND,cc,SHA1) / E3(WPA2,SHA1,cc)|0.0
E2(WPA2,cc,SHA1,PMS-) / E3(WPA2,SHA1,cc)|0.0
E2(WPA2,cc,SHA1,PMSE) / E3(WPA2,SHA1,cc)|0.0
E2(WPA2,cc,SHA1,P---) / E3(WPA2,SHA1,cc)|0.0
E2(WPA2,ct,MD5) / E3(WPA2,MD5)|0.0
E2(WPA1,tt,MD5) / E3(WPA2,MD5)|0.0
E2(WPA2,tt,MD5) / E3(WPA2,MD5)|0.0
E2(WPA2,ct,MD5,PMSE) / E3(WPA2,MD5)|0.0
E2(WPA2,tt,MD5,PMSE) / E3(WPA2,MD5)|0.0
E2(WPA2,tc,SHA1) / E3(WPA2,SHA1,cc)|0.0
CCMP_DATA / CCMP_DATA
BRD / Beacon+CCMP_Broadcast
+CCMP_Multicast
BRD / Beacon+
Unencrypted_Multicast
Authenticated
s3
E4(WPA2,SHA1) / CCMP_DATA|0.0
E4(WPA1,MD5) / CCMP_DATA|0.0
E4(WPA1,MD5,PM--) / CCMP_DATA|0.0
E4(WPA2,MD5) / CCMP_DATA|0.0
E2(WPA2,cc,SHA1) / CCMP_DATA|0.0
E2(WPA1,cc,SHA1) / CCMP_DATA|0.0
E2(RAND,cc,SHA1) / CCMP_DATA|0.0
E2(WPA2,tc,SHA1) / CCMP_DATA|0.0
E2(WPA2,ct,MD5) / CCMP_DATA|0.0
E2(WPA1,tt,MD5) / CCMP_DATA|0.0
E4(RC=W) / CCMP_DATA|0.0
E4(NONC=W) / CCMP_DATA|0.0
DELAY / Disas|6.0
DATA / Disas|6.0
CCMP_DATA / Disas|6.0
E2(MIC=F) / Disas|0.0
E2(MIC=F,SHA1,P---|) / Disas|0.0
E2(MIC=F,SHA1,PMSE|) / Disas|0.0
E4(MIC=F) / Disas|0.0

Figure 5.3: State machine for the TP-Link TL-WR841HP, with normal transitions highlighted in green and those contributing to vulnerabilities in red. Non-vulnerable implementation specific transitions are denoted in black.

out a 4-Way Handshake with the target AP, using AES-CCMP as the selected cipher. Messages are selectively forwarded and altered between the target AP and client. Message 1 will contain the same nonce (for generation of the session key), but will be altered such that the cipher suite flag is set to TKIP. The client will generate its own nonce, calculate the session key, then send a TKIP MICed Message 2, which will be forwarded unchanged to the AP. This is accepted by the target and induces a downgrade to TKIP, resulting in a TKIP protected Message 3 response. The attacker will then observe this message, and can extract the RC4 protected GTK. The encrypted key could then be recovered and used for various attacks (see [117]).

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

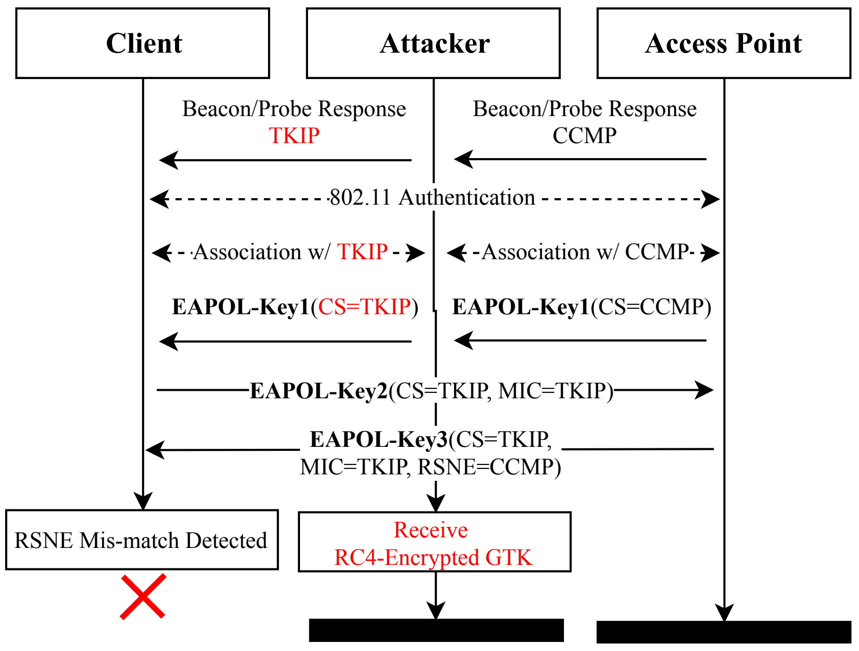

Client
Attacker
Access Point
Beacon/Probe Response TKIP
Beacon/Probe Response CCMP
802.11 Authentication
Association w/ TKIP
Association w/ CCMP
EAPOL-Key1(CS=TKIP)
EAPOL-Key1(CS=CCMP)
EAPOL-Key2(CS=TKIP, MIC=TKIP)
EAPOL-Key3(CS=TKIP, MIC=TKIP, RSNE=CCMP)
RSNE Mis-match Detected
Receive RC4-Encrypted GTK

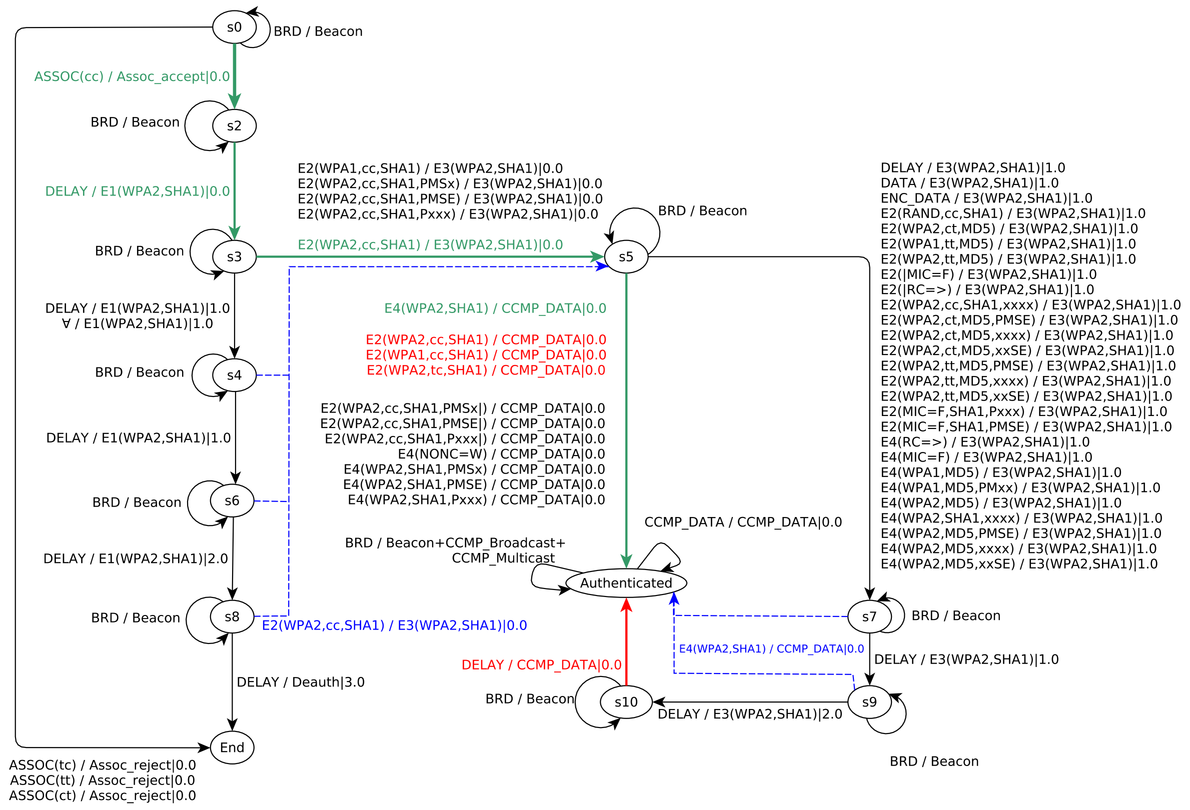
Figure 5.4: Downgrade attack on the TP-Link WR841P to force encryption of the Group Key (GTK) with RC4

of TKIP. Normally, the client will always choose the more secure AES-CCMP.

With the Cisco WAP121, this vulnerability is indicated by the fact that Message 4, the message sent by the client to confirm the selected cipher, is not required by the AP. We can see from the state machine diagram Figure 5.5, after 3 re-transmissions of Message 3 over a period of 4s , the AP will give up waiting for a response and complete the connection anyway. Exploitation of this vulnerability is illustrated in Figure 5.6.

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

For both APs, this attack is limited to downgrading the AP only. Correctly implemented clients will detect this downgrade through an inconsistency of the RSNE infor-



s0
BRD / Beacon
ASSOC(cc) / Assoc_accept|0.0
s2
BRD / Beacon
DELAY / E1(WPA2,SHA1)|0.0
s3
BRD / Beacon
E2(WPA1,cc,SHA1) / E3(WPA2,SHA1)|0.0
E2(WPA2,cc,SHA1,PMSx) / E3(WPA2,SHA1)|0.0
E2(WPA2,cc,SHA1,PMSE) / E3(WPA2,SHA1)|0.0
E2(WPA2,cc,SHA1,Pxxx) / E3(WPA2,SHA1)|0.0
E2(WPA2,cc,SHA1) / E3(WPA2,SHA1)|0.0
s5
BRD / Beacon
DELAY / E1(WPA2,SHA1)|1.0
∀ / E1(WPA2,SHA1)|1.0
s4
BRD / Beacon
E4(WPA2,SHA1) / CCMP_DATA|0.0
E2(WPA2,cc,SHA1) / CCMP_DATA|0.0
E2(WPA1,cc,SHA1) / CCMP_DATA|0.0
E2(WPA2,tc,SHA1) / CCMP_DATA|0.0
E2(WPA2,cc,SHA1,PMSx|) / CCMP_DATA|0.0
E2(WPA2,cc,SHA1,PMSE|) / CCMP_DATA|0.0
E2(WPA2,cc,SHA1,Pxxx|) / CCMP_DATA|0.0
E4(NONC=W) / CCMP_DATA|0.0
E4(WPA2,SHA1,PMSx) / CCMP_DATA|0.0
E4(WPA2,SHA1,PMSE) / CCMP_DATA|0.0
E4(WPA2,SHA1,Pxxx) / CCMP_DATA|0.0
DELAY / E1(WPA2,SHA1)|1.0
s6
BRD / Beacon
DELAY / E1(WPA2,SHA1)|2.0
BRD / Beacon+CCMP_Broadcast+
CCMP_Multicast
CCMP_DATA / CCMP_DATA|0.0
Authenticated
s8
BRD / Beacon
E2(WPA2,cc,SHA1) / E3(WPA2,SHA1)|0.0
DELAY / Deauth|3.0
End
ASSOC(tc) / Assoc_reject|0.0
ASSOC(tt) / Assoc_reject|0.0
ASSOC(ct) / Assoc_reject|0.0
DELAY / CCMP_DATA|0.0
BRD / Beacon
s10
DELAY / E3(WPA2,SHA1)|2.0
E4(WPA2,SHA1) / CCMP_DATA|0.0
s7
BRD / Beacon
DELAY / E3(WPA2,SHA1)|1.0
s9
BRD / Beacon
DELAY / E3(WPA2,SHA1)|1.0
DATA / E3(WPA2,SHA1)|1.0
ENC_DATA / E3(WPA2,SHA1)|1.0
E2(RAND,cc,SHA1) / E3(WPA2,SHA1)|1.0
E2(WPA2,ct,MD5) / E3(WPA2,SHA1)|1.0
E2(WPA1,tt,MD5) / E3(WPA2,SHA1)|1.0
E2(WPA2,tt,MD5) / E3(WPA2,SHA1)|1.0
E2(|MIC=F) / E3(WPA2,SHA1)|1.0
E2(|RC=>) / E3(WPA2,SHA1)|1.0
E2(WPA2,cc,SHA1,xxxx) / E3(WPA2,SHA1)|1.0
E2(WPA2,ct,MD5,PMSE) / E3(WPA2,SHA1)|1.0
E2(WPA2,ct,MD5,xxxx) / E3(WPA2,SHA1)|1.0
E2(WPA2,ct,MD5,xxSE) / E3(WPA2,SHA1)|1.0
E2(WPA2,tt,MD5,PMSE) / E3(WPA2,SHA1)|1.0
E2(WPA2,tt,MD5,xxxx) / E3(WPA2,SHA1)|1.0
E2(WPA2,tt,MD5,xxSE) / E3(WPA2,SHA1)|1.0
E2(MIC=F,SHA1,Pxxx) / E3(WPA2,SHA1)|1.0
E2(MIC=F,SHA1,PMSE) / E3(WPA2,SHA1)|1.0
E4(RC=>) / E3(WPA2,SHA1)|1.0
E4(MIC=F) / E3(WPA2,SHA1)|1.0
E4(WPA1,MD5) / E3(WPA2,SHA1)|1.0
E4(WPA1,MD5,PMxx) / E3(WPA2,SHA1)|1.0
E4(WPA2,MD5) / E3(WPA2,SHA1)|1.0
E4(WPA2,SHA1,xxxx) / E3(WPA2,SHA1)|1.0
E4(WPA2,MD5,PMSE) / E3(WPA2,SHA1)|1.0
E4(WPA2,MD5,xxxx) / E3(WPA2,SHA1)|1.0
E4(WPA2,MD5,xxSE) / E3(WPA2,SHA1)|1.0

Figure 5.5: Learned model for the Cisco WAP121. Note that for retransmission states, assumed transitions (as per Assumption 3) are represented by dotted blue lines.

mation which is selected in the Association stage and later encrypted and encapsulated within Message 3 from the AP. The client will decrypt the contents of the message, find that in fact the AP supports AES-CCMP and should then drop the connection.

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

Disclosure TP-Link and Cisco were fully informed of the vulnerabilities found, and in line with responsible disclosure, were given 6 months to address the vulnerabilities before publication. Cisco assigned CVE-2018-0412 for the affected WAP121 access point and a

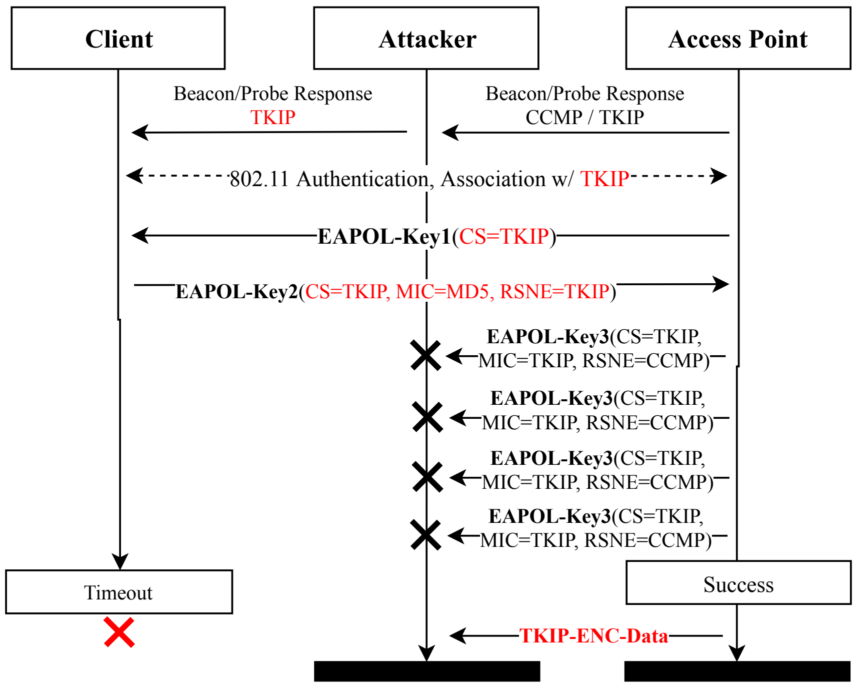

Client
Attacker
Access Point
Beacon/Probe Response
TKIP
Beacon/Probe Response
CCMP / TKIP
802.11 Authentication, Association w/ TKIP
EAPOL-Key1(CS=TKIP)
EAPOL-Key2(CS=TKIP, MIC=MD5, RSNE=TKIP)
EAPOL-Key3(CS=TKIP, MIC=TKIP, RSNE=CCMP)
EAPOL-Key3(CS=TKIP, MIC=TKIP, RSNE=CCMP)
EAPOL-Key3(CS=TKIP, MIC=TKIP, RSNE=CCMP)
EAPOL-Key3(CS=TKIP, MIC=TKIP, RSNE=CCMP)
Timeout
Success
TKIP-ENC-Data

Figure 5.6: Downgrade attack on the Cisco WAP121 to force usage of TKIP

number of other affected models. We additionally note that TP-Link no longer sell the vulnerable AP.

5.5 Discussion and Limitations

We have shown that model learning algorithms, traditionally designed for fully deterministic systems, can be adapted to learn lossy protocols such as WPA/2. We have also presented an efficient and complementary approach to learn typical time-behaviour defined by protocol implementations.

As a direction of future work, we would like to apply our technique to more protocols that are either lossy, time is important, or both. Our technique of time learning separation would be particularly suited to other protocols where long timeouts are present, making standard learning difficult to use. There are many security protocols where timing plays an important role, especially those running on unreliable mediums, such as other wireless protocols (Bluetooth, Zigbee, LTE) or distance-bounding protocols (MasterCard's RRP, NXP's "proximity check" [36]). Indeed, recent work (as of 2020) on learning DTLS [53] implementations cited the similar challenges to those tackled in this work. As this protocol is designed to to run on UDP based connections, included in it's function is the retransmission of handshake messages. The authors of [53] note that these are a major cause of non-determinism, so various tricks are deployed in order to avoid triggering them. This therefore is an ideal future candidate for testing our approach.

In order to define our proposed approach, and understand the scenarios it is applicable to, we proposed a number of assumptions about typical protocol behaviour. Inevitably, these assumptions introduce potential limitations. One limitation of our approach comes from Assumption 1, which states that no simultaneous clocks must be in operation. Whilst we have not seen evidence of this assumption being violated in security protocol handshakes, one could imagine protocol implementations which have both state specific timeouts in addition to a global timeout for the whole handshake. Moreover, a variation of the selective-repeat data transmission protocol which includes multiple timers has been proposed [64].

With respect to our proposed approach in dealing with non-determinism, recent work from 2019 [100], on learning a state machine of the QUIC protocol [76], identified a case of inconsistent responses to a particular query. A majority-vote based variant of our approach was attempted, however did not prove to resolve the issue. The root cause of this was due to the inconsistent query-response in question carrying over data from previous handshake attempts—meaning that a majority response could not be identified with a stable, majority probability. This highlights the fact that our approach requires full and complete state resets, which for the example of learning a particular implementation of QUIC was not possible.

Part III

Grey-box Implementation Analysis

Chapter Six

Grey-Box Protocol State Machine Learning

In this chapter we tackle the analysis of protocol implementations from a unique angle. We first highlight the inherent constraints of black-box based analysis, which includes both the type of behaviour that can be inferred, and the limited insights for post-learning model examination. We then proceed to propose an alternative approach to alleviate these issues. In particular, we take a modest 'peek' inside the box, and combine typical black-box model learning algorithms with the monitoring of runtime memory combined with novel static binary analysis. We implement our approach and demonstrate its efficacy by testing a number of implementations of the real-world security protocols TLS and WPA/2. We additionally provide case studies of protocols for which black-box learning fails, but our method learns correctly.

6.1 Motivation

Current approaches to learn state machines for widely used protocol implementations are fundamentally limited. In this section, we demonstrate the need for a new direction; we present three problems that highlight the deficiencies in current approaches, and then present a corresponding set of research questions that serve as guiding principles for the design and evaluation of our improved method.

Problem 1: Black-box approaches are unable to effectively learn protocol state machines with *deep* state changes.

Consider the state machine in Fig. 6.1. In this simple protocol, a *trapdoor* exists that allows us to transition from state 1 (not authenticated) to state 2 (authenticated) by supplying an input of 11 `init` messages, as opposed to an `auth` message[1].

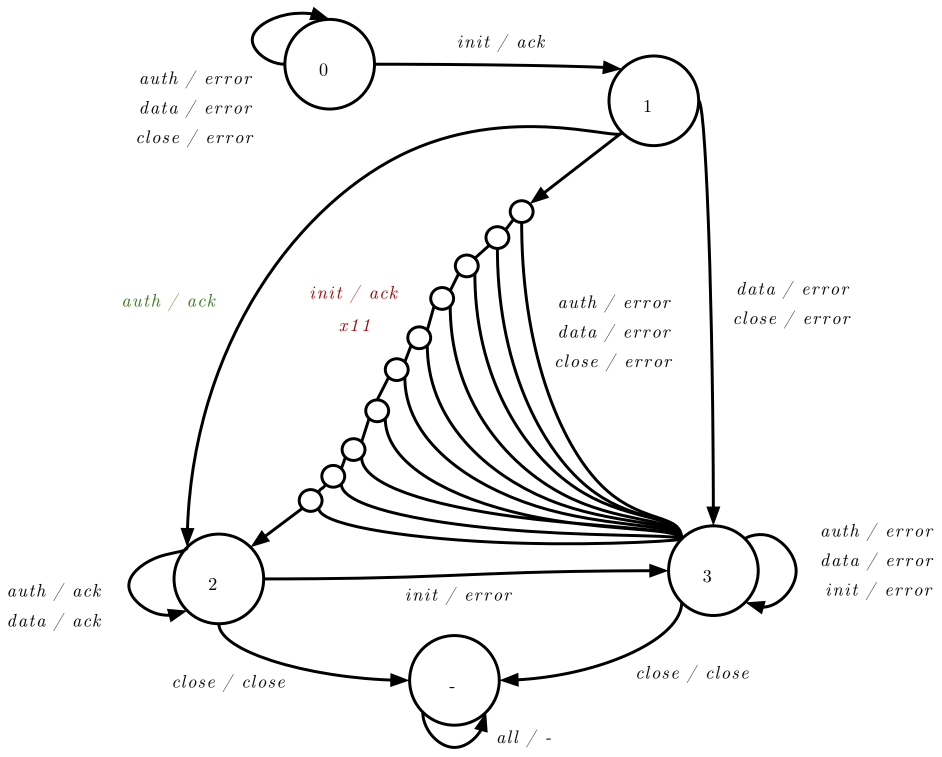


Figure 6.1: A 13 state deep protocol state machine with a backdoor that can be activated by repeated attempts of the *init* input from state 1.

To learn this state machine, state-of-the-art model learning with the TTT-Algorithm

[1]This backdoor is similar to the vulnerability presented in Chapter 5, Figure 5.5, which permitted a cipher downgrade after failed 3 failed steps of the handshake. Only with a depth equivalence checking depth of 3+ was this reachable using black box methods.

[68] and modified W-Method [44] requires around 360k queries to discover the backdoor, but fails to terminate after 1M queries—essentially operating by brute-force. This is because the number of queries posed is polynomial in the number of messages, states and exploration depth. Moreover, to identify this backdoor, we must set a maximum exploration depth of at least 11. In practice, however, since query complexity explodes with a high bound, most works use a much lower bound (e. g. 2 in [44, 54]), or use random testing which lacks completeness. This query explosion is also exacerbated by large input sets. Hence, for systems where time-spent-per-query is noticeable (e. g. a few seconds, as is the case with some protocols), it is apparent that learning such systems requires a more efficient approach.

Problem 2: Black-box approaches are limited to insights derived from I/O queries.

The test harness which interacts with the SUL (System Under Learning) necessarily makes assumptions about how the SUL works internally, for example, how nonces and flags are set and reset, for each protocol message, and how any ambiguities in a protocol standard are handled. If these assumptions are incorrect, then an incorrect state machine will be produced. Since black-box learning is limited to a view of only I/O behaviour, identifying the cause of such incorrectness is difficult, as we cannot easily discern between a bug in the implementation, or an incorrect assumption in the test harness, without both expert knowledge and manual analysis of the harness and the implementation—drastically diminishing the advantages of automated learning.

In the paper by de Ruiter et al. [44], the authors acknowledge that the state machines presented for OpenSSL 1.0.1g and 1.0.1j are incorrect due to differing assumptions between the implementation and their test harness. The incorrectness stems from a specific test-harness configuration, resulting in some states not being modelled correctly. For complex protocols, implementing a test harness is a significant engineering challenge, and black-box learning, by nature cannot help to discern between bugs in the harness and bugs in the SUL, due to its limited insights—raising the potential for such bugs to go unnoticed, since we must rely on expert domain knowledge and inspection of each implementation by hand.

Problem 3: Black-box approaches cannot observe how and when a SUL interacts with its environment.

Without any insight into how a SUL processes its input, to make learning tractable, black-box approaches bound their exploration using a conservative depth limit. This imposes a practical limit on the type of bugs that can be discovered, and the complexity of protocols that can be learnt with respect to the size of the input message set. It also means that black-box approaches often do no better than brute-force: they can neither optimise the number of queries they make, nor detect if a SUL has stopped processing new input. Rather, they continue to pose queries *blindly* until their exploration bound is reached.

Unless the SUL is truly a black-box to the analyst orchestrating the learning process, then it is almost always possible to gain additional insight into how it executes, aside from just I/O behaviour. We can exploit this fact to substantially improve the learning process. By monitoring both the memory of a SUL and how it performs I/O, we can effectively *focus* learning and pinpoint where we should explore to greater depths and when we should terminate learning early. For example, when certain exploration paths and their corresponding execution paths produce no effect on state-defining memory locations, we can identify this and terminate exploration, without resorting to querying to an a priori defined bound. In fact, it is the case that such insights are beneficial for all protocols we test. As an exceptional example, for the TLS implementation of RSA BSAFE for C, our approach is able to identify that input is no longer read by the SUL upon reaching an error state, and thus can terminate learning at this point. In contrast, black-box learning [44] cannot use this insight and so continues exploration beyond the error states until its exploration bound is reached, resulting in tens of thousands of superfluous queries.

Clearly, when applying state machine learning to analyse real-world protocol implementations, by not looking *inside* the black-box, we limit the type of protocols that can be learnt and the type of vulnerabilities that can be discovered. Further, when looking inside the box, we must carefully decide which properties to observe, as attempting to learn a complete input-to-state correspondence, using, e. g. symbolic execution, will suffer from state explosion, and simply replace the exploration bound of black-box learning with a symbolic exploration bound.

6.2 Contribution

In this chapter we present a new paradigm in model learning. Our technique examines behaviour "under-the-hood", with respect to its effect on program execution state. The implication of this is that we can learn models that capture more behaviour and gain insights beyond input/output behaviour for model inspection and understanding.

Our state machine learning method works in two stages. In the first stage, we capture snapshots of the implementation's execution context (i. e. memory) whilst it runs the protocol, which we then analyse to identify the locations in memory that define the protocol state for each possible state. In the second stage, we learn the complete state machine by sending inputs to the protocol implementation and analysing the identified locations to see which values the state defining memory takes, and so which protocol state has been reached. This allows us to recognise each protocol state as soon as we reach it, making learning much more effective than previous methods that require substantially more queries.

Similar to black-box learning, we are able to reason about the correctness of our approach, and provide guarantees about the state machines it is able to learn. We are able to enumerate the assumptions we make succinctly, and verify that they are both reasonable and realistic for analysing implementations of complex, widely deployed protocols, including TLS and WPA/2. Further, we are able to demonstrate case studies of protocols that contain difficult to learn behaviour, that our method is able to learn correctly, but state of the art black box approaches cannot learn: either due to non-termination, or because they output the incorrect state machine.

- We present the design and implementation of a new model inference approach, StateInspector, which leverages observations of run-time memory and program analyses. Our approach is the first which can map abstract protocol states directly to concrete program states.
- We demonstrate that our approach is able to learn states beyond the reach of black-box learning, while at the same time improve learning efficiency. Further, the models we produce allow for novel, post-mortem memory analyses not possible with other approaches.
- We evaluate our approach by learning the models of widely used TLS and WPA/2 implementations. Our findings include a high impact vulnerability, several concerning deviations from the standards, and unclear documentation leading to protocol misconfigurations.

6.3 Overview and Definitions

We depict a high-level overview of our approach in Figure 6.2. Our method combines insights into the runtime behaviour of the SUL (provided by the execution monitor and concolic analyser) with observations of its I/O behaviour (provided by the test harness) to learn models that provide comparable guarantees to traditional black-box approaches. We organise this section by first stating the assumptions we make about the state machines

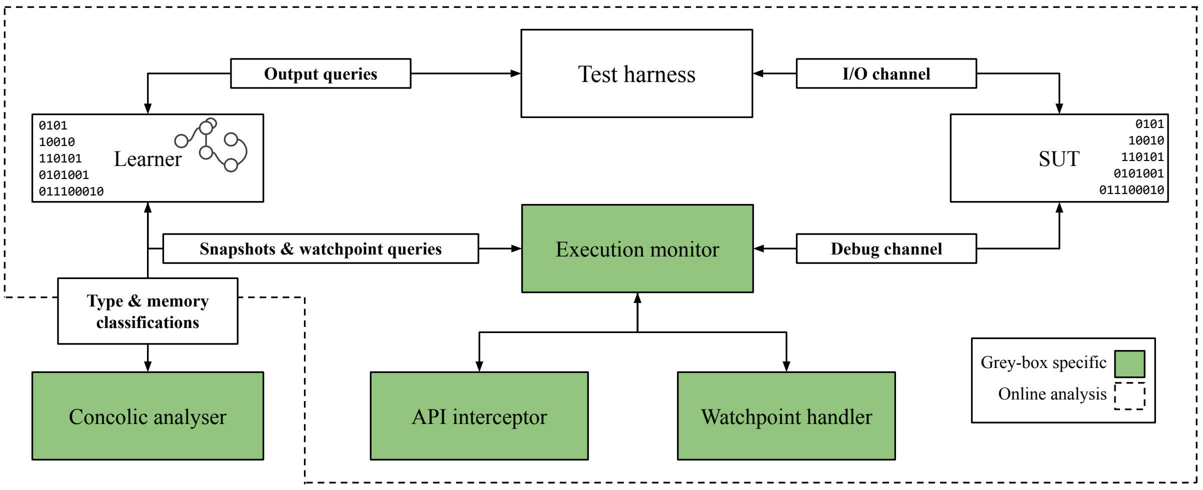


Figure 6.2: Grey-box State Learning Architecture.

our approach learns, and then provide an overview of its operation. So that the reader has a clear understanding of all assumptions we make of the implementations we analyse, we also state assumptions about the configurable parameters of our method. We defer discussion of the choice of their concrete assignments, and how the assumptions can be loosened (and the implications of doing so) to later sections, so that they can be understood in context.

Our first assumption is standard for state machine learning:

Assumption 1. *The protocol state machine of the SUL is finite, and can be represented by a Mealy machine.*

In all concrete implementations, each state of this Mealy machine will correspond to a particular assignment of values to specific "state-defining" memory locations, allocated by the implementation. We use two terms to discuss these locations and the values assigned to them, which we define as follows:

Definition 1 (Candidate State Memory Location) *Any memory location that takes the same value after the same inputs, for any run of the protocol.*

Definition 2 (A Set of State-Defining Memory) *A minimal subset of candidate state memory locations whose values during a protocol run uniquely determine the current state.*

For simplicity, we also refer to a member of any state-defining memory set as *state memory*. Our next assumption is that we know a priori the complete language needed to interact with the SUL and hence generate the state machine:

Assumption 2. *All protocol states can be reached via queries built up from the inputs known to our testing harness I^+.*

Under these assumptions, we now describe the operation of our approach. Our learner uses a test harness to interact with the SUL. This test harness is specific to each protocol analysed and tracks session state, and is responsible for communicating with the SUL and translating abstract representations of input and output messages (for the learner) to concrete representations (for the SUL). In the first phase of learning, the learner instructs the test harness to interact with the SUL to exercise normal protocol runs, trigger errors, and induce timeouts multiple times. We capture snapshots of the SUL's execution context (i. e. its memory) for each of these runs using an execution monitor. We perform subsequent analysis of the snapshots in order to determine a set of candidate memory locations that can be used to reason about which state the SUL is in. We make the following assumption about this memory:

Assumption 3. *A set of state-defining memory for all states is allocated along a normal protocol run, and this memory will take a non-default value in the normal run, during error handling, or timeout routines.*

Any memory location found to have the same value for each sequence of inputs for each run is considered candidate state memory. And any location that takes the same value in all states is discarded. Assumption 3 ensures that this reduced set of locations will be a set of state-defining memory, however, this set may also contain superfluous locations. Further, this set of locations may not be state-defining for every state, and is rather a superset of locations for all states.

The next phase of learning uses the candidate set to construct a model of the protocol state machine. During this process, we identify the set of state-defining memory for each state. This allows us to determine if two states arrived at from the same input sequence should be considered equal, even if their memory differs, as we only need to consider the values of state-defining locations when performing this so-called equivalence check. It also allows us to effectively ignore any superfluous locations in our candidate set since they will not be considered state-defining for any state.

To learn the state machine, we queue all queries from I^+ to fulfil the *completeness* property for states in a Mealy machine. Namely, all inputs are defined for each state. We perform these queries iteratively in increasing length. As in the first phase, we take snapshots of the execution state of the SUL on each I/O action. Each time a distinct state, according to its I/O behaviour and defined state memory, is discovered, we queue additional queries to ensure that the state is also fully defined (i. e. we attempt to determine all states reachable from it). We continue building the automaton in this way until no new states are found.

As well as state memory, it is possible that our candidate set contains locations with assignments that appear to be state-defining for all states identified from the set of posed inputs, but are actually not. For instance, if a protocol implementation maintains a counter of the messages received, it might be mistaken for state memory, leading to a self loop in the state machine being conflated with an infinite progression of seemingly different states. Recognising such memory will prevent our framework from mistaking a looping state from an infinite series of states, and so ensures termination. To determine if such memory exists and if it can be ignored, we perform a *merge check*. This allows us to replace a series of states repeated in a loop, with a single state and a self loop. This check is only performed between states that have the same observed input and output behaviour, and are connected at some depth (i. e. one is reachable from the other). We make the following assumption about this depth:

Assumption 4. *The depth that we check for possible merge states is larger than the length of any loop in the state machine of the implementation being tested.*

When performing a merge check, we consider two states equal if the values assigned to their state-defining locations are equal. Therefore, for each location that differs, we must determine if it behaves as state memory for each state (we discuss the conditions for this in Section 6.4.5). To do so, we use a novel analysis (implemented in the concolic analyser in Figure 6.2). This analysis identifies a memory location as state-defining when this location influences decisions about which state we are in. Concretely, this is encoded by checking if the location influences a branch that leads to a write to any candidate state memory. That is, state-defining memory will control the state by directly controlling writes to (other) state memory, and non-state-defining memory will not. This leads to our final assumption:

Assumption 5. *Any state-defining memory will control a branch, along an execution path triggered by messages from the set I^+, and it will lead to a write to candidate state memory within an a priori determined number of instructions.*

We discuss the selection of the above bound in Section 6.4.5, but note that it is a configurable parameter of our algorithm. We complete learning when no further states can be merged, and no new states (that differ in I/O behaviour or state memory assignment) can be discovered by further input queries. At this point, StateInspector outputs a representative Mealy machine for the SUL.

6.4 Methodology

The learner component of Figure 6.2 orchestrates our model inference process. It learns how the SUL interacts with the world by observing its I/O behaviour via the test harness, and learns how it performs state-defining actions at the level of executed instructions and memory reads and writes, via the execution monitor. Inference begins with a series of bootstrap queries in order to identify candidate state memory, i. e. M. All queries posed to the SUL in this stage serve the dual purpose of identifying state memory and also refining the model. This ensures minimal redundancy and repetition of queries. Given an estimation of M, we then proceed to construct the state machine for the implementation by identifying each state by its I/O behaviour and set of state-defining memory. Throughout this process we perform *merge checks* which determine if two states are equivalent with respect to the values of their state-defining locations.

6.4.1 Monitoring Execution State

To monitor a SUL's state while it performs protocol related I/O, we take snapshots of its memory and registers at carefully chosen points. As many protocols are time-sensitive, we ensure that this monitoring is lightweight to avoid impacting the protocol's behaviour. More specifically:

1. We avoid inducing overheads such that the cost of each query becomes prohibitively high.
2. We do not interfere with the protocol execution, by, for example, triggering timeouts.
3. We only snapshot at points that enable us to capture each input's effect on the SUL's state.

To perform a snapshot, we momentarily pause the SUL's execution and copy the value of its registers and contents of its mapped segments to a buffer controlled by the execution monitor. To minimise overheads, we buffer all snapshots in memory, and only flush them to disk after the SUL has finished processing a full query. We find that the overheads induced by snapshotting in this way are negligible and do not impact the behaviour of any of the protocols analysed.

To identify when to perform snapshots, we infer which system calls (syscalls) and fixed parameters a SUL uses to communicate with its client/server. We do this by generating a log of all system calls and passed parameters used during a standard protocol run. We then identify syscall patterns in the log and match them to provided inputs and observed outputs. The number of syscalls for performing I/O is small, hence the number of patterns we need to search for is also small, e. g. combinations of `bind`, `recvmsg`, `recvfrom`, `send`, etc. When subsequently monitoring the SUL, we hook the identified syscalls and perform state snapshots when they are called with parameters matching the those in our log.

Mapping I/O Sequences To Snapshots

To map I/O sequences to snapshots, we maintain a monotonic clock that is shared between our test harness (which logs I/O events) and our execution monitor (which logs snapshot events). We construct a mapping by *aligning* the logged events from each component, as depicted in Figure 6.3. When doing so, we face two key difficulties. First, implementations may update state memory before or after responding to an input, hence we must ensure we take snapshots to capture both possibilities (i. e. case ❸ in Figure 6.3). Second, some parts of a query may not trigger a response, hence we must account for the absence of *write*-like syscalls triggering a snapshot for some queries (i. e. cases ❷ and ❹).

For all scenarios, we trigger *input* event snapshots after *read*-like syscalls return, and *output* event snapshots just before *write*-like syscalls execute. We take additional

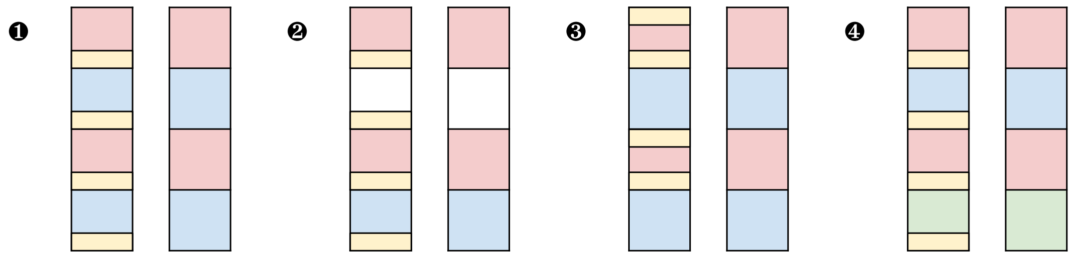


Figure 6.3: Mapping I/O sequences to snapshots. We depict four cases ❶-❹, snapshot events are shown on the left and I/O on the right. We indicate read/input events in red, write/output events in blue, state changes in yellow, and connection close events in green. ❶ shows the case where snapshots and state changes are trivially aligned with I/O. ❷ shows the case where a state change occurs without a corresponding output event. ❸ shows the case where state changes occur after output events. ❹ shows the case where a state change occurs after the last input event, with no output before the connection is closed.

snapshots for *output* events before each *read*-like syscall. This enables us to always capture state changes, even if no corresponding *write*-like syscall occurs, or the state change happens after the output is observed.

If there is no output to the final input of a query q, then we instruct the learner to pose an additional query with an extra input, $q||i$ for all $i \in I$. We expect there to be a *read* event corresponding to the final input of one of these queries—thus providing us with a snapshot of the state after processing the penultimate input. If the SUL does not perform such a read, i. e. it stops processing new input, as in case ❹, we detect this by intercepting syscalls related to socket closure, and inform the learner that exploration beyond this point is unnecessary.

6.4.2 Identifying Candidate State Memory

In the first stage of learning, our objective is to identify a set of possible memory locations whose values represent state (Definition 1), i. e. M. Our analyses aim to find M such that the only locations it contains are values that can be used to discern if we are in a given state, for all possible protocol states that the SUL can be in.

Snapshot Generation To form an initial approximation of M, we perform bootstrap queries against the SUL, this produces a set of memory snapshots where all of the memory which tracks state is defined and used (Assumption 3). Our bootstrap queries take the form, $BF = \{b_0, b_1, ...b_n\}$, where $b_i \in I^+$. The first of these queries b_0 is set as the *happy flow* of the protocol. This is the expected normal flow of the protocol execution, which we assume prior knowledge of. For example, in TLS 1.2, $b_0 = (ClientHelloRSA, ClientKeyExchange, ChangeCipherSpec, Finished)$. The other queries in BF are specific mutations of this happy flow, automatically constructed with the intention of activating error states, timeout states, and retransmission behaviour. Each of these queries b_x is derived by taking all prefixes p_x of the happy flow b_0, where $0 < |p_x| \leq |b_0|$, and for all inputs in I, and appending to p_x each $i \in I$ a fixed number of times T such that $b_x = p_x || i^T$.

Every bootstrap query is executed at least twice, so that we have at least two snapshots for each equivalent input sequence, which we require for the next step of our analysis. When compared to black-box approaches, one might assume that our method requires many more queries in order to facilitate learning. This is not the case. In fact, once we identify a SUL's state memory, we can reuse all of the bootstrap queries for refining our model in subsequent phases.

When possible, we also attempt to alternate functionally equivalent input parameters to the SUL across bootstrap flows. We do this to maximise the potential number of locations we eliminate due to not holding the same value at equivalent states (Assump-

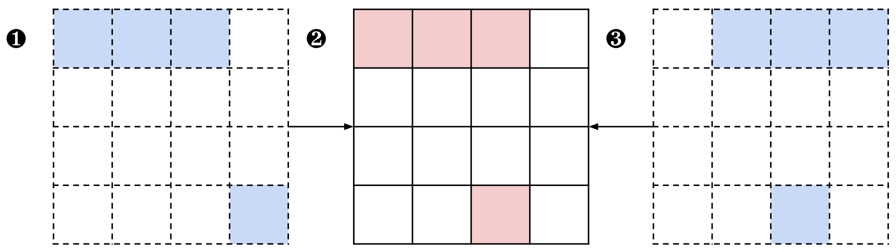


Figure 6.4: Allocation alignment across different executions. Each large block represents the memory layout of a different protocol run; coloured squares represent state-defining locations. Allocation alignment computes a mapping of allocations of a given run (blue squares) to a *base* configuration (red squares). Using this mapping, we can *diff* snapshots that have different configurations, e. g. ❶ and ❸, by first mapping them onto a common configuration, i. e. ❷.

tion 3). For example, for TLS implementations, we execute some bootstrap flows with different server certificates, which enables us to eliminate memory that would otherwise have to be identified as non-state-defining using our concolic analyser—a comparatively heavyweight procedure.

Handling Dynamic Memory To facilitate handling multiple simultaneous sessions, client/server implementations generally allocate state memory on-demand. This presents a challenge when identifying state-defining locations, as logically equivalent allocations may not reside at the same offsets across different protocol executions. To address this non-determinism, we compute a mapping of each execution's allocations to a single *base* configuration. This enables us to analyse all snapshots together with respect to a common configuration, rather than pair-wise. Figure 6.4 visualises our approach. We first construct an allocation log for each execution that records a timestamp, the stack pointer (i. e. callee return address), and call parameters, of every call to a memory allocation function (e. g. `malloc` and `free`). Then, to derive a mapping between two logs, we *align* their allocations/frees, by matching them on their log position, allocation size, and calling context (recorded stack pointer). We choose our *base* configuration, or log, as the largest *happy flow* log, under the assumption that it will contain all allocations related to state-defining locations for any possible session.

Snapshot Diffing Following mapping each bootstrap snapshot's allocations onto the *base* log, we *diff* them to obtain our candidate set M. Each element $m \in M$ corresponds to the location of a contiguous range of bytes of dynamically allocated memory, which we represent by: an allocation address, an offset relative to the start of the allocation, and a size.

We perform diffing by first grouping all snapshots by their associated I/O sequences. Then, for each group, we locate equivalent allocations across snapshots and identify allocation-offset pairs which refer to byte-sized locations with the same value. We then check that every identified location also contains the same value in every other I/O equivalent snapshot, and is a non-default value in at least one snapshot group (Assumption 3). This gives us a set of candidate state memory locations. We note that as this process is carried out at the *byte* level, we additionally record all bytes that do not abide by this assumption. This enables us to remove any misclassified bytes once we have established the real bounds of individual locations (Section 6.4.3).

6.4.3 Minimising Candidate State Memory

Given our initial set of candidate state memory locations, we reduce M further by applying the following operations:

1. **Pointer removal:** we eliminate any memory containing pointers to other locations in memory. We do this by excluding values which fall within the address-space of the SUL and its linked libraries.
2. **Static allocation elimination:** we remove any full allocations of memory which are assigned a single value in the first snapshot which does not change throughout

the course of our bootstrap flows.

3. **Static buffer elimination:** we remove any contiguous byte ranges larger than 32 bytes that remain static.

Static allocation and buffer elimination are used to filter locations corresponding to large buffers of non-state-defining memory, for example, in OpenSSL, they eliminate locations storing the TLS certificate.

Candidate State Memory Type-Inference

The values stored at state-defining locations are often only meaningful when considered as a group, e. g. by treating four consecutive bytes as a 32-bit integer. Since we perform snapshot diffing at a byte-level granularity, we may not identify the most-significant bytes of larger integer types if we do not observe their values changing across snapshots. As learning the range of values a location can take is a prerequisite to determine some kinds of termination behaviour, we attempt to monitor locations with respect to their intended types.

To learn each location's bounds, we apply a simple type-inference procedure loosely based upon that proposed by Chen et al. [31]. We perform inference at the same time as we analyse snapshots to handle uncertain state memory locations (Section 6.4.5). During this analysis, we simulate the SUL's execution for a fixed window of instructions, while doing so, we analyse loads and stores from/to our candidate state memory. To perform inference, we log the prefixes used in each access, e. g. in `mov byte ptr [loc_1], 0`, we record `byte` for `loc_1`. Then, following our main snapshot analysis, we compute the maximal access size for each location and assign each location a corresponding type. To disambiguate overlapped accesses, we determine a location's type based on the minimal non-overlapped range. For example, if we observe that `loc_1` and `loc_1+2` are both accessed as 4-byte

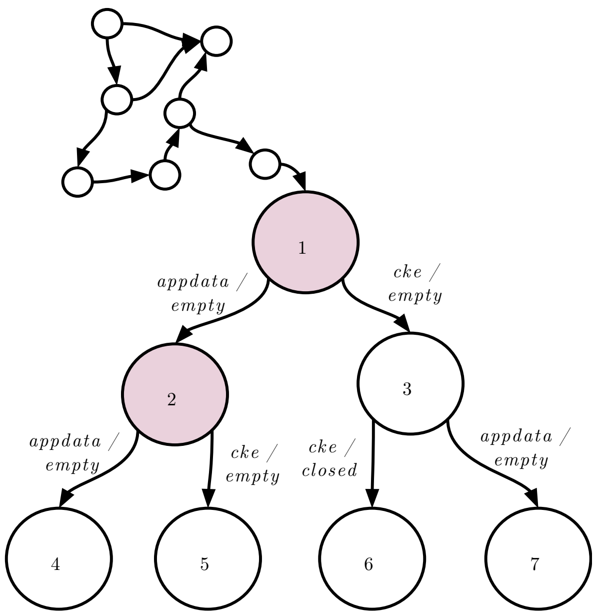


Figure 6.5: An example scenario where merge conditions for states 1 and 2 are met, when D is set to 1. The state learner would as such taint test any differing memory between the two states to estimate if this memory is state defining.

values, we compute `loc_1`'s type as two bytes and `loc_1+2` as four bytes. Following type-inference, we update our model and the state classifications maintained by our learner using the new type-bounds discovered.

6.4.4 State Merging

We consider a unique assignment of candidate state memory a unique state (Definition 2). Hence two states reachable by the same inputs, with equivalent assignments for these locations can be considered equal and merged. However, by applying a trivial equality check, if M is not minimal, i. e. it is an over-approximation, this check may not yield the correct outcome. Which, in the worst case, could lead to non-termination of learning. We therefore must address this to ensure learning is possible in all cases with respect to Assumption 1, namely that the state machine being learned is finite. The method we present in this section handles this possibility by allowing for states to be merged under the assumption that our candidate set contains superfluous locations. We attempt to merge states in the model each time all queries in the queue of a given length have been performed. We call this the *merge check*. It operates by identifying pairs of *base* and *merge* states, which can be merged into a single state. These pairs are selected such that two properties hold: ❶ the *merge* state must have *I/O equivalence* to the *base*, and ❷ the *merge* state should be reachable from the *base*. I/O equivalence signifies all input-output pairs are equal for the two states, to a depth D. The intuition is that if I/O differences have not manifested in D steps, they may in fact be the same state, and we should therefore check if the differences in their memory are state relevant. The base-to-merge state reachability must also be possible in D steps (Assumption 4). We enforce this property based on the observation that in security protocol implementations, state duplication is more likely with states along the same *path*. So-called *loopback* transitions often occur where inputs are effectively ignored with respect to the state, however their processing can still result in some changes to memory, resulting in duplication for our memory classified states. Loops in models between multiple states are less common, but will be checked by our learner provided the number of states involved is less than or equal to D.

In Figure 6.5, we present an example of where the mergeability properties hold, with $D = 1$ for states 1 and 2—they are I/O equivalent to depth 1, and state 2 is reachable from state 1. Our merge check determines whether the memory that distinguishes the states is state-defining. Algorithm 1 shows a sketch of our approach. In summary, for each location of differing memory, and for each input message (to ensure completeness), we monitor the execution of the SUL. We supply it with input messages to force it into the state we wish to analyse (i. e. state 1 or 2 in our example in Figure 6.5), we then begin to monitor memory reads to the tracked memory location. We follow this by supplying the SUL with each input in turn and perform context snapshots on the reads to the memory location, which we call a *watchpoint hit*. We then supply each of these snapshots as inputs to our concolic analyser (detailed in the next section), which determines if the SUL uses the value as state-defining memory (Assumption 5). If our analysis identifies any location as state defining, then we do not perform a merge of the two states. Conversely, if all tested locations are reported as not state-defining, then we can merge the *merge* state into the *base* state, and replace the transition with a self loop.

Algorithm 1: Generation of watchpoint hit snapshots.

```
    Result: list of $watchpointHitExecutionDumps$
    Input: $(addr, size)$ of memory to test, prefix query $p$ to arrive at
           $mergeState$, the $SUL$
1   foreach $(input, st) \in reachableFrm(mergeState, 1)$ do
2       if $outputFor(input, st) \in disabled$ then
3           $continue$;
4       end
5       $initialiseWatchpointMonitor(SUL, addr, size)$;
6       $executeQuery(SUL, p||input)$;
7   end
```

6.4.5 Handling Uncertain State Memory

Forming our candidate set M by computing the differences between snapshots gives us an over-approximation of all of the locations used discern which state the SUL is in. However, an implementation may not use all of these locations to decide which state it is in for every state. As described in the previous section, when testing if two memory configurations for the same I/O behaviour should be considered equivalent, we must identify if differing values at candidate locations imply that the configurations correspond to different states (i. e. the differing locations are state-defining), or if the values they take are not meaningful for the particular state we are checking. For a given state, by analysing how locations actually influence execution, we can determine state-defining locations from those that are not. This is because in all cases, non-state-defining locations will have no influence on how state is updated. However, since this kind of analysis is expensive, the diffing phase is crucial in minimising the number of candidate locations to analyse—typically reducing the number to tens, rather than thousands.

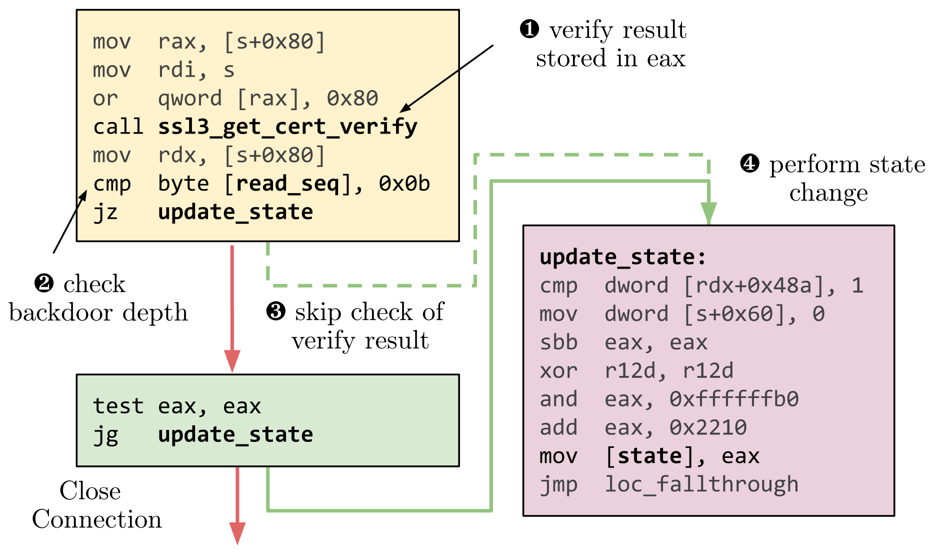
Properties Of State-Defining Memory

To confirm a given location is state-defining, we attempt to capture execution traces of its location behaving as state-defining memory. We summarise these behaviours below, which we base on our analysis of various implementations:

1. **Control-dependence:** writes to state memory are control-dependent on reads of state memory. To illustrate this, consider the typical case of a state enum flag read forming the basis of a decision for which code should process an incoming message, and the resulting state machine transition defined by a write to state memory.
2. **Data-dependence:** non-state memory that is conditionally written to due to dependence on a read from state memory may later influence a write to state memory.
3. **State-defining and state-influential locations:** for a particular state, its state-defining memory locations (Definition 2) will always be read from before they are written to (to tell which state one is in), and subsequent writes to state memory will be control- or data-dependent upon the values of those reads (Assumption 5). For state-influential locations, e. g. input buffers, this property will not hold—while the contents of a buffer may *influence* state, it will not directly *define* it, and will be written to before being read.

Discerning State-Defining Locations

If a location holds more than one value when performing a merge check, we apply an additional analysis to determine if it is state-defining. First, we identify execution paths



```
mov  rax, [s+0x80]
mov  rdi, s
or   qword [rax], 0x80
call ssl3_get_cert_verify
mov  rdx, [s+0x80]
cmp  byte [read_seq], 0x0b
jz   update_state
```

```
test eax, eax
jg   update_state
```

```
update_state:
cmp  dword [rdx+0x48a], 1
mov  dword [s+0x60], 0
sbb  eax, eax
xor  r12d, r12d
and  eax, 0xffffffb0
add  eax, 0x2210
mov  [state], eax
jmp  loc_fallthrough
```

Figure 6.6: OpenSSL client verification bypass. At ❶ we check the client certificate signature, at ❷ we check if the `read_seq` counter is equal to 11, if so, we bypass the check of the result of client certification verification via ❸. At ❹ we perform a state update that is control-dependent on `read_seq`'s value. StateInspector identifies the dependence and dynamically adjusts the learning depth to discover the deep-state change.

that are control-dependent on the value stored at the location. Then, we check if any of those paths induce a write to known state memory. If so, we classify the location as state-defining.

We base our analysis on a variation of byte-level taint propagation combined with concolic execution. We apply it in sequence to state snapshots taken on reads of the candidate location *addr*. We start analysis from the instruction *pc* performing the read that triggered the state snapshot. Our analysis proceeds by tainting and symbolising *addr*, then tracing forwards until we reach a branch whose condition is tainted by *addr*. If we do not reach a tainted conditional within W instructions, we do not continue analysing the path. At the conditional, we compute two assignments for the value at *addr*—one for each branch destination. We then symbolically explore each of the two paths until we have processed W instructions or we reach a return instruction. If we observe a write to known state memory on either path, we consider *addr* to be state-defining. Figure 6.6 depicts an example location that requires such analysis. The counter `read_seq` is used to implement a sneaky backdoor that bypasses client certificate verification in OpenSSL. Our analysis finds that `read_seq` taints the branch `jz update_state`, and thus, leads to a state change. We therefore classify it as state-defining.

We select an analysis bound of $W = 512$ based on the observation that reads from memory involved in comparisons tend to have strong locality to the branches they influence. Since we analyse multiple state snapshots for each *addr*, generated for every possible input in I, we reduce the chance for missing locations used in state-defining ways outside of our bound. In practice, we find that our selection of W leads to no false negatives, and note that it can be configured to a larger value to handle problematic implementations.

6.4.6 Implementation

Our resulting implementation consists of a Java-based learner, a ptrace-based [20] execution monitor, and test harnesses in Python and Java. Our concolic analyser is built using Triton [98] and IDA Pro [65]. In total, StateInspector consists of over 10kloc across three different languages.

6.5 Results

In this section we present the evaluation of our approach, StateInspector, in which we test a number of implementations of two security protocols, TLS and WPA/2. We additionally carry out tests on a number of purpose built protocols, where we have deliberately planted *deep* states.

Our experiments show that our approach makes it possible to identify the presence of *deep* states (e. g. in our purpose built protocols), and enhance the performance of learning with large input sets (e. g. WPA/2). In order to illustrate the additional insights over I/O that our tool affords, we discuss our findings related to two OpenSSL implementations. In particular, we discover that despite I/O differences between some states, we find that at the memory level, they are equivalent. Finally, we evaluate StateInspector against the state-of-the-art in black-box learning algorithms. We show that in all cases, our approach learns models in less queries and provides better termination conditions, and in all but one case, is capable of learning models in less time. For the case where our method is slower, we purposely bound the learning to take as much time as black-box learning, so we can then fairly compare the accuracy of the resulting state machines.

Table 6.7: Model learning results on TLS 1.2 and WPA/2 servers. TLS experiments are repeated with two alphabets, core and extended. Times specified in hours and minutes (hh:mm). Various query counts are abbreviated to Qs in column headings. Experiments which did not terminate with in 3 days are denoted: ■.

	Proto.	Implementation	Classifying Mem.		Mem.	I/O	Total	I/O Mem.	Watchpoints		Total	Black-Box	
			Locs.	Allocs.	States	States	Qs.	Qs.	Qs.	Hits	Time	Qs.	Time
Core Alpha.	TLS	RSA-BSAFE-C	88	14	11	9	128	109	19	4	00:06	204k	■
	TLS	Hostap TLS	159	32	11	6	194	160	34	17	00:24	971	01:48
	TLS	OpenSSL 1.0.1g	126	16	19	12	488	280	208	116	00:18	5819	00:42
	TLS	OpenSSL 1.0.1j	125	16	13	9	291	165	126	23	00:10	1826	00:13
	TLS	OpenSSL 1.1.1g	126	6	6	6	88	86	2	0	00:02	175	00:02
	TLS	GnuTLS 3.3.12	172	7	16	7	265	129	36	42	00:10	1353	00:21
	TLS	GnuTLS 3.6.14	138	7	18	8	973	452	522	121	00:36	3221	00:52
Ext. Alpha.	TLS	RSA-BSAFE-C	165	7	14	7	918	688	230	635	01:08	133k	■
	TLS	OpenSSL 1.1.1g	467	66	11	11	535	524	11	0	01:05	1776	01:25
	TLS	OpenSSL 1.0.1g	554	110	40	20	2454	1816	594	609	04:21	20898	21:05
	TLS	GnuTLS 3.3.12	185	5	23	19	1427	1274	153	128	02:05	66k	■
	TLS	GnuTLS 3.6.14	157	6	13	11	730	673	57	37	01:32	66k	■
	WPA/2	Hostap 2.8†	138	3	24	6	1629	804	825	261	03:30	2127	03:30
	WPA/2	IWD 1.6	135	8	5	5	264	126	138	597	01:02	3000+	08:00+

† For Hostap, we stopped both learners at around 200 minutes, as we found that its state machine is infinite and would prevent both approaches terminating.

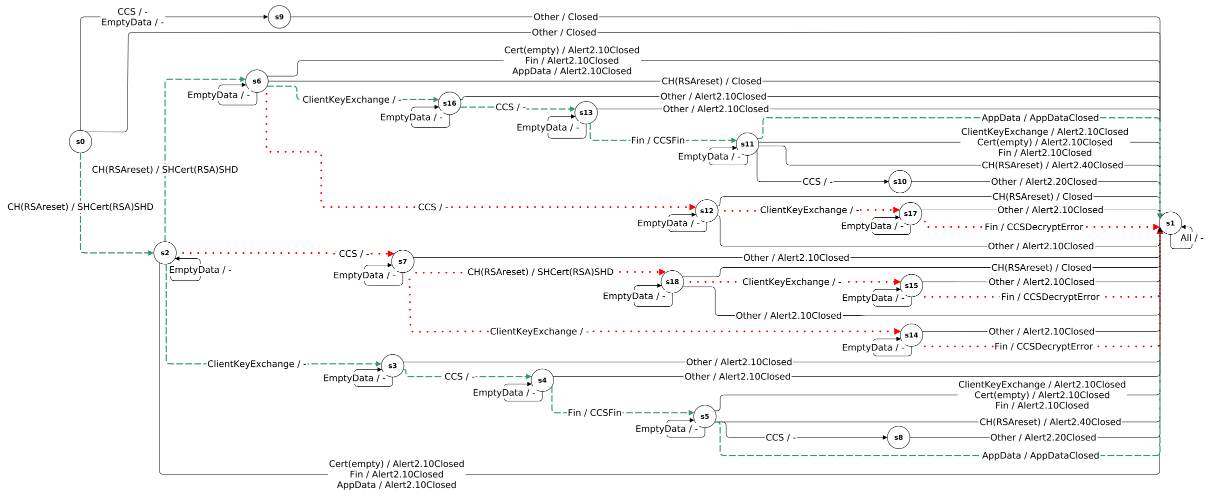


Figure 6.8: Learnt model for OpenSSL 1.0.1g

6.5.1 TLS 1.2

We evaluate the interesting examples of TLS 1.2 servers from the work of de Ruiter et al. [44], the latest versions of these servers, and both the server and client implementations of Hostap's internal TLS library. This spans 4 different code bases, and 7 unique implementations. We learn these models with two separate input sets, one containing the core TLS messages (as in de Ruiter et al.), and where possible, another with a much larger input set, including client authentication and multiple key exchanges (as provided by the TLS-Attacker test harness [110]). Table 6.7 lists the time taken to learn all models, as well as the identified *candidate state memory* details, query statistics and number of states identified. Further, we list the number of queries required to learn the same models with state-of-the-art black-box learning, i. e. the TTT algorithm [68] with the modified W-Method equivalence checking of [44] with a conservative equivalence checking depth of 3. For practical reasons, we cut off black-box experiments after 3 days if they show no sign of approaching termination.

OpenSSL

We tested a number of different versions of OpenSSL, ranging from 2014 to 2020. Of particular interest is the OpenSSL server running version 1.0.1g. A state machine for this implementation was presented at Usenix 2015 [44]. It modelled a previously discovered critical vulnerability: specifically, if a `ChangeCipherSpec` (`CCS`) message was sent before the `ClientKeyExchange`, the server would compute the session keys using an empty master secret. Prior to finalising our model, which we depict in Figure 6.8, we replicated the notably different model presented in [44], by using an identical test harness. As described in Section 6.1, this harness sent invalid `Finished` messages on paths with more than one `ClientHello`. This results in a model where the dashed green path (s0, s2, s6, s16, s13, s11) and dotted red path (s0, s2, s7, s18, s15) in Figure 6.8 always lead to an `Alert` and connection closure after receiving the final input. As this was a test harness configuration issue, we sought to use StateInspector in order to confirm this fact. In Table C.1 (Appendix C), we list the differences in memory between states on alternative paths to handshake completion, and the known happy flow state s4. These differences were identified once learning had terminated. As merge conditions were never met for these states, we carried out a post-analysis stage, where we used StateInspector to determine whether the differing memory defined the respective states. We found that it did not. Despite these states sharing different I/O definitions, when compared to state s4, all were equivalent in terms of state-defining memory, thus indicating that the test harness was responsible for the differences. This example also illustrates where I/O interpretation by the test harness is misleading. States s15 and s17, which manifest due to the aforementioned early-`CCS` vulnerability, are also equivalent to the happy flow state s4. The I/O difference in this case is due to the fact that the test harness cannot decrypt the response (hence `DecryptError`). We note that this post-learning analysis does not come for free, therefore, we only execute such checks when further model investigation is needed.

When testing client authentication with the extended alphabet, we also discovered inconsistencies in the documentation, which, at worst, have the potential to cause vulnerable misconfigurations. We discuss this further in Appendix C.

RSA-BSAFE-C

We selected this implementation as it serves to demonstrate that our method is applicable not just to open-source implementations, but also to closed-source. The implementation demonstrates a further advantage of our approach over black-box learning. In particular, we observe that each time the server sends an `Alert` message, it performs a partial socket shutdown. Specifically, it executes a `shutdown(n, SHUT_RD)` on the connection with the test harness. This means that it no longer accepts input, however, this is not detectable by the test harness. For our approach, this is not a problem; we can observe the `shutdown` syscall and prevent further inputs from this point. For black-box learning, the socket closure is not detected, and so learning continues exploring beyond the receipt of an `Alert` message. As shown in Table 6.7, this leads to many superfluous queries, and, as a result, the learner fails to terminate within 3 days for either alphabet. In comparison, our algorithm is able to learn the same models with 128 queries in 6 minutes for the core TLS functionality, and in ~1 hour for a more complex model capturing client authentication and alternative key exchanges. Notably, this latter test revealed that repeated `ClientHello`'s are only permitted when the server is configured with forced client authentication.

GnuTLS

Our tests on the TLS server implementations in GnuTLS 3.3.12 and 3.6.14 showed substantial changes between the two versions. In particular, each version required us to hook different syscalls for snapshotting. State count also differed, especially so when testing with the extended alphabet. Analysis of the models revealed slightly different handling of Diffie-Hellman key exchanges, which in the older version resulted in a path of states separate from RSA key exchange paths.

The difference in learning performance between the two approaches, in the case of the extended alphabet, was profound. Black-box learning failed to terminate after 3 days and over 60k queries. We found that this was due to multiple states and inputs warranting empty responses. Consequently, black-box learning exhausted its exploration bound, trying all possible combinations of the troublesome inputs at the affected states. In contrast, as shown in Table 6.7, StateInspector is able to handle such cases much more effectively. This is in part because some groups of inputs are found to result in equivalent snapshots, and when the state equivalence is not immediately evident, our merging strategy quickly finds memory differences are inconsequential.

Hostap-TLS

We tested Hostap's internal TLS library as both a client and server. Although this library is described as experimental, it is used in resource-constrained Wi-Fi devices with limited flash space [39] and by Wi-Fi clients in embedded Linux images created using Buildroot [27], e. g. motionEyeOS [41].

Surprisingly, we found that this TLS library always sends TLS alerts in plaintext, which might leak information about the connection. Further, some frames from the extended alphabet, such as Heartbeat requests or responses were not supported, and sending them resulted in desynchronisation of the TLS connection. We therefore do not include learning results for this implementation with the extended alphabet.

More worrisome, the model showed that against a client, the `ServerKeyExchange` message can be skipped, meaning an adversary can instead send `ServerHelloDone`. The client will process this message, and then hits an internal error when sending a reply because no RSA key was received to encrypt the premaster secret. When Ephemeral

Diffie-Hellman is used instead, the client calculates g^{cs} as the premaster secret with s equal to zero if no `ServerKeyExchange` message was received. Because the exponent is zero, the default math library of Hostap returns an internal error, causing the connection to close, meaning an adversary cannot abuse this to attack clients. Nevertheless, this does illustrate that the state machine of the client does not properly validate the order of messages.

6.5.2 WPA/2's 4-Way Handshake

We also tested WPA/2's 4-way handshake which is used to authenticate with Wi-Fi networks. There are two open source implementations of this handshake on Linux, namely the ones in IWD (iNet Wireless Daemon) and Hostap, and we test both.

To learn the state machine, we start with an input alphabet of size 4 that only contains messages which occur in normal handshake executions. Our test harness automatically assigns sensible values to all of the fields of these handshake messages. To produce larger input sets, we also tried non-standard values for certain fields. For example, for the replay counter, we tried the same value consecutively, set it equal to zero, and other variations. To this end, we created two extended alphabets: one of size 15 and another one of size 40.

To also detect key reinstallation bugs [118], we let the SUL send an encrypted dataframe after completing the handshake. In the inferred model, resulting dataframes encrypted using a nonce equal to one are represented using `AES_DATA_1`, while all other dataframes are represented using `AES_DATA_n`. A key reinstallation bug, or a variant thereof, can now be automatically detected by scanning for paths in the inferred model that contain multiple occurrences of `AES_DATA_1`.

One obstacle that we encountered is handling retransmitted handshake messages that were triggered due to timeouts. Because this issue is orthogonal to evaluating our memory-based state inference method, we disabled retransmissions, and leave the handling of retransmissions as future work.

IWD

We found that the state machine of IWD does not enter a new state after receiving message 4 of the handshake. This means an adversary can replay message 4, after which it will be processed by IWD, triggering a key reinstallation. Note that this is not a default key reinstallation, i. e. we confirmed that existing key reinstallation tools cannot detect it [116]. The discovered reinstallation can be abused to replay, decrypt, and possibly forge data frames [118]. We reported this vulnerability and it has been patched and assigned CVE-2020-17497. When running our tool on the patched version of IWD, the state machine enters a new state after receiving message 4, which confirms that the vulnerability has been patched.

With black-box testing, the key reinstallation is only found when using a small input alphabet *and* when the test harness always sends handshake messages with a replay counter equal to the last one used by the AP. If the harness instead increments the replay counter after sending each handshake message, the key reinstallation can only be discovered when using an extended alphabet. However, in that case black-box learning takes much longer: after 3 hours the learner creates a hypothesis model that includes the vulnerability, but it is unable to verify this hypothesis within 8 hours (at which point we terminated the learner). This shows that our method handles larger input alphabets more efficiently, especially when queries are slow, resulting in more accurate models and increasing the chance of finding bugs.

Hostapd

One obstacle with Hostapd is how the last used replay counter in transmitted handshake messages is saved: our learner initially includes this counter in its candidate state memory set. When trying to merge states with different replay counter values, our concolic analysis determines this value to be state defining. Indeed, Hostapd checks whether incoming frames include a particular value. However, with each loop of the state machine, the expected value changes, and hence the learner prevents the merge. This results in a violation of Assumption 1, meaning learning will not terminate. Like with other violations of assumptions, we addressed this by including a time bound on the learning. Because of this we also do not explore a bigger alphabet. For a fair comparison, we used the same time bound in our method and in black-box learning. The models which were produced with both approaches within ~200 minutes were equivalent. Under these conditions, we note that both resulting state machines followed the standard and contained no surprising transitions.

6.5.3 Example Protocols

In order to highlight the capability of StateInspector to identify states deep in a protocol, we test a minimally modified version of OpenSSL, in addition to a variety of example protocols. We note that states in protocols beyond typical learning bounds are not only theoretical—a state machine flaw in a WPA/2 router [81] found a cipher downgrade was possible after 3 failed initiation attempts (which would have been missed with configured bounds less than 3, as in [44]).

Our modification of OpenSSL (version 1.0.1j), consists of a 4 line addition (Appendix C, Listing C.1), which hijacks existing state data to implement a simple client authentication bypass, activated by n unexpected messages (see Figure 6.6 of Section 6.4.5). To learn this model, we used an extended alphabet of 21 messages, including core TLS functionality, client certificates, and various data frames. With $n = 5$, black-box learning configured with a bound equal to n, fails to identify the state after 3 days of learning and ~100k queries (and showed no signs of doing so soon). StateInspector, on the other hand, identifies the main backdoor-activation state in under 2 hours and ~1.2k queries and the second activation variant (with two preceding `ClientHellos`, as opposed to the usual one) in under 3 hours and ~2.5k queries. Doubling the depth n results in 3.2k and 4k queries respectively—an approximately linear increase. Black-box learning predictably failed to locate the even deeper state due to it's exponential blow up in this particular scenario.

Our motivating example protocol of Section 6.1, Figure 6.1 also showed a dramatic advantage over black-box learning, even with a simple model and small input alphabet. We implemented this protocol similarly to a real protocol, with state maintained between an enum and counting integer combined. With black-box learning, this model took 360k queries and 20 hours to identify the auth-bypass but failed to terminate its analysis after 4 days. StateInspector identified the backdoor in 149 queries in just over minute, with the concolic analyser taking up 20 seconds of that time.

6.6 Discussion and Limitations

We have shown that, in many ways, relaxing the restriction of pure black-box analysis allows for improved model learning results: new types of states can be learnt, more insights can be gained, all in less time. While we have demonstrated our assumptions make a justifiable trade-off between practical usability and completeness, we acknowledge that they may however not hold for all protocols.

If the machine is not finite (Assumption 1), then, like black-box methods without out bound, our approach will fail to terminate. Likewise, with respect to Assumption 4, if loops within the state machine exist beyond the configured depth, and the memory across these states is always changing (e. g. a counter), then StateInspector will interminably duplicate states. This scenario is also possible if an attempted state merge operation fails, which may for example happen if our taint-analysis based dependency test returns a false positive (ref. Section 6.4.5). In all of these cases, like black-box methods, we provide a means to bound exploration. We support options to do this either by configurable time, or by an exploration depth. Since our bound is enforced at exploration from a specific troublesome state, this remains a substantial improvement over black-box learning which performs bounded exploration from *every state*. In the worst case, the resulting models will be no less representative than those learnt with black-box methods.

False negatives with respect to candidate state memory identification and concolic analysis also have the potential to be problematic. A consequence of Assumption 3 is that we may miss candidate state memory as a result of not observing a change from its initialised value within the bootstrap flows. We note that it is possible to implement a protocol for which this assumption fails, for instance having a number of error states that are defined by memory that is only allocated after an error occurs, or representing the state as, say, the difference between two pieces of memory, rather than their absolute value. However, our investigation of code examples and our results indicate that this assumption does hold for the majority of protocols.

An inherent requirement of grey-box analysis is the ability to introspect a program's execution state. Whilst we ensure this is performed as noninvasively as possible, it does limit the types of SUL which can be tested. For example, our method would not be appropriate for protocols running on certain embedded devices (e. g. EMV cards). However, for devices with sufficient debug access (e. g. JTAG), our method can be deployed and allows implementations to be learnt where black-box learning proves to be too slow.

Finally, although StateInspector is designed to work with both closed and open-source implementations, it is often the case—as with most of our tested protocols—that we do have access to source code. If this is the case, our tool is able to map uses of state memory to source code locations—which substantially aids in the analysis of inferred models.

Part IV

Closing Statements

Chapter Seven

Conclusion

Throughout the course of this thesis we have explored a variety of approaches aimed at uncovering potential logic flaws in protocol implementations. We began with an approach for identifying a specific vulnerability (missing hostname verification) in a specific protocol (TLS) in a specific setting (mobile apps). The technique we developed, although applied in a focused domain, was founded upon the general observation that program properties can be inferred by observing outputs corresponding to given inputs. This is a particularly fruitful approach with input/output systems like protocols, where most program behaviour is observable by simply monitoring the network.

Exploring this line of thinking further, we adopted a procedure for inferring a model of a protocol's implemented behaviour through active automata learning. Like with the work of Chapter 1, this approach consisted of dynamically driving the execution of a target protocol via manipulated inputs. In contrast however, the approach was much more systematic. The resulting protocol state machine model provided a high level abstraction of the protocol functionality which proved to be straight forward to analyse and identify suspect behaviour. This was not without its challenges—adaptations were required for our case study (WPA/2), as non-deterministic output observations, combined with time-triggered behaviour, both prevented straight forward application of existing learning algorithms, and hindered learning time-complexity.

In the final chapter of this thesis, we experimented with a fundamentally different technique for learning the effect that given inputs have on the state of a program. We identified inherent limitations with state modelling based on inputs and outputs alone, and instead designed a tool which leverages a view of the programs runtime memory. Compared with the *black-box* approach of Chapter 5, here we operated within a *grey-box* framework. This represents a parting from the model learning work of the past two decades, which we argued is approaching its limit of effectiveness. We implemented our grey-box approach and tested it on a variety of security protocols. Our experiments found this to be a promising technique that not only expands the class of behaviour that can be learned, particularly in the case of hard to reach states, but also improves on general learning performance - in some cases by an order of magnitude.

The work carried out in this thesis has brought to light a number of possible future directions for this line of research. In the next section, we discuss this in detail, including both improvements to the techniques presented in this thesis, as well as important orthogonal problems in the domain of protocol model learning.

7.1 Future work

TLS in mobile apps Our certificate verification and pinning analysis of Chapter 4, of what were mostly high-security mobile apps, exposed the continued difficulty in developing correct and secure implementations. At the time of this research, Google had begun to roll out a new API (Network Security Configuration) which intended to make certificate verification easier and implementations more homogenous. However, as of 2021, and 4 years after our study was carried out, Oltregge et al. [91], revealed that despite large-scale adoption of this new API, mis-configurations and other implementation difficulties remained. This highlights the fact that security testing in this area remains important. The study was however limited to a single API for implementing certificate pinning, in part due to the fundamentally limited static analysis approach that is taken (parsing configuration files). On the other hand, our approach allowed for an implementation (or API) independent testing solution, which was however much more restricted with respect to the number of apps that could be tested (∼400 vs. ∼1 million apps). The reason for this comparatively low number of tested apps was due to the requirement to manually interact with apps in order to trigger runtime behaviour (TLS connection attempts). As such, a useful line of future work would be automated emulation of apps, combined with the network-level classification and detection of particular bug classes (like pinning without hostname verification). This would enable a much wider perspective of the ecosystem to be obtained.

Model learning with time Chapter 5 of this thesis presented an approach for learning time-triggered behaviour in protocols. We tested this approach on implementations of the WPA/2 4-way handshake, and found that in most cases learning was significantly sped up, and in others, a necessity. This achievement however, required enforcing assumptions which may not hold for all protocols and their implementations. Our first assumption limited the number of clocks to one. Tackling the problem of learning with multiple clocks has been addressed in numerous studies, however much work remains to bring current ideas into practice. Recently (as of 2020), other researchers have also identified the need of learning algorithms which account for restricted single timer systems, and developed formal algorithms to learn their models [11, 94]. Unlike our algorithm, these approaches do not make any assumptions regarding retransmissions (ref. Assumption 2 and 3 in Section 5.3.2), and as such future comparative and experimental work would be beneficial. Within the domain of security protocol testing, combining this with work of Fiterau-Brostean et al. on DTLS model learning [53], would also constitute an ideal case study for future experiments, as DTLS also defines time behaviour (e. g. timeouts and retransmissions).

Finally, modelling in the presence of time behaviour is also a challenge for future work with respect to the StateInspector approach of Chapter 6. State related to time is often not visible within memory, rather it is computed dynamically based on the difference between e. g. the current time, and a given time stamp taken at protocol initialisation. A future StateInspector style approach may be possible through a tighter integration to black-box learning algorithms, especially those focused on time (such as that presented in Chapter 5, or those in the aforementioned recent works).

State memory accuracy in StateInspector The state-memory based model learning technique of Chapter 6 relies on the accuracy of identified state memory for the correctness of learned models. Whilst we have not yet empirically detected modelling errors due to incorrect memory identification, we envisage improvements to this algorithm in future work. Improving this aspect of the algorithm primarily revolves around type inference (or bound identification) of single variables (or units) of state memory. As state memory is located through "diffing" at the byte-level, it is possible that we miss the full range of values that single variables can take. We partially address the problem through a primitive form of type inference based on the size of memory accesses (ref. Section 6.4.3), however this will not suffice for complex data types. Previous works, most notably Howard [109], proposed more advanced forms of memory access analysis for dynamic "excavation" of data structures like struct and array types. Adoption of a technique like this, integrated into active learning, may therefore prove fruitful. An alternative approach may be to leverage white-box compiler pass techniques, like those used by Poeplau et al. [97] for symbolic execution. In this way, we could in effect *log* type information of data used throughout execution, and proceed to match up these types with the candidate state memory locations. This would require source-code access (i. e. white-box analysis), however, as we have seen, most widely used application level security protocols have open-source implementations. As well as improving correctness of models, these aforementioned techniques will also aid learning efficiency. This is because state merging complexity (Section 6.4.4) is proportional to the number of single state-memory units to test. Improved knowledge of unit bounds means that less memory can be treated as individual bytes. Consequently, merges will require fewer queries that gather watchpoint hits which are used to determine if differing memory is state defining.

The implication of better state memory identification improving the efficiently of learning relates to another avenue for future work. In particular, our preliminary experiments indicated the potential for StateInspector's ability to efficiently learn with large input sets. This, however, would benefit from further experimental evidence (for example, testing large input sets in TLS with the TLS-Attacker test harness [110]), in combination with future algorithmic improvements to reduce false positives in state memory identification.

Automatic input set inference One of the major limiting factors in automatic protocol model learning is that of the fixed input set. Any states present in a target implementation which cannot be reached with only inputs of this set, can not be learned. Protocols with a small input space can be learned by enumerating all possible values of input messages. This, however, quickly becomes infeasible with any protocol which has inputs of size greater than a few bytes, since learning complexity is polynomial in the size of the input set. Model learning would therefore benefit from the ability to learn *inputs* that can refine a state machine model, just like *states* are learned. As we discussed in our review of related work (Section 3.3.2), this challenge has previously been tackled from two different angles—symbolic execution in MACE from 2011, and fuzzing in the more recent works by Janssen et al. [70] and AFLnet [95] from 2020. Both works are promising but have yet to be successfully applied to security protocols like those covered in this thesis. Notably, these solutions suffer from the limited state information provided by inputs/outputs. That is, any newly discovered inputs (through symbolic execution or fuzzing) must be tested, in the typical black-box way, against the target to determine whether the corresponding output sequence refines the model. As we know from the work of Chapter 6 however, this may not be sufficient to learn the effect of inputs on state, and therefore runs a significant risk of prematurely discarding inputs which may refine a model at different point. We believe the memory-based state classification of StateInspector may aid such input-discovery techniques for model learning, since the memory view of state can detect state influential inputs which are not evident in immediate I/O.

Appendices

Appendix One

Downloaded HSBC Configuration Data

```
...
<supportedDevices>
<device name="iPhone"/>
<device name="iPad">
<configurl>https://www.hsbc.co.uk/content_static/
tablet/1/5/17/1/config.json</configurl>
</device>
</supportedDevices>
<supportedSSLPinning>
<enable>false</enable>
</supportedSSLPinning>
<localised locale="en" name="United␣Kingdom" shortname="UK" default="true"/>
<configurl>https://www.hsbc.co.uk/content_static/mobile/1/5/17/1/config.json</
    configurl>
</entity>
<entity id="144" headerImg="default" hometype="app">
<supportedDevices>
<device name="iPhone"/>
<device name="iPad">
<configurl>https://mapp.us.hsbc.com/1/PA_1_083Q9FJ08A002FBP5S00000000/
content/usshared/Mobile/tablet/HBUS_1-5-TabletApp_Jun2015_PROD.js</configurl>
</device>
</supportedDevices>
...
```

Listing A.1: XML config file downloaded over vulnerable HSBC connections.

Appendix Two

Learning Correctness Probabilities

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

$$correct(i) = {}^nC_i p_e^i (1-p_e)^{n-i} = \frac{n!}{i!(n-i)!} p_e^{n-i} (1-p_e)^i$$

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

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

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

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

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

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

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

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

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

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

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

Appendix Three

OpenSSL Analysis With StateInspector

Table C.1: The sets of differing memory for states on suspected and confirmed alternate paths to successful handshake completion, when compared to the legitimate *happy flow* state 4 (s4) memory.

State IDs	Addresses of differing memory to State 4
s14	{0x555555a162a0, 0x4b0, 0x13}
	{0x555555a162a0, 0x4b0, 0x120}
	{0x555555a162a0, 0x4b0, 0x0}
	{0x555555a0e6b0, 0x160, 0x0}
	{0x555555a0e6b0, 0x160, 0x10}
s15, s17, s13	{0x555555a0e6b0, 0x160, 0x44}
	{0x555555a0e6b0, 0x160, 0x68}
	{0x555555a0e6b0, 0x160, 0x6c}
	{0x555555a0e6b0, 0x160, 0xc0}
	{0x555555a0e6b0, 0x160, 0xc8}
	{0x555555a0e6b0, 0x160, 0xd2}

Using the learning output of StateInspector, we carried out an investigation into what the differing memory in Table C.1 actually constituted. We were able to do this by referring to the logs produced by the tool, which print the source code details of any watchpoint hits made at reads of the tested memory. All considered states shared the two differing addresses listed at the top of the table. The first of these represented a counter referred to as the `sequence number`. This value increments for each message received after the `ChangeCipherSpec` message and is used for constructing MACs. The second address in the table refers to the frame type of the just received message. This value is determined as not state defining because from any given state it is always written to before being read, i. e. it may be state influencing, but not defining (ref. 6.4.5). The third address in the table pertains to a flag which determines if the client has attempted more than one `ClientHello` message. Only prior to the receipt of a `ClientKeyExchange` is this flag ever read (and therefore used to define the state). Consequently, it is not considered state defining for any of the states listed in the table. The remaining memory refers to arbitrary certificate data which we also found was not state defining. In particular, we were unable to detect any reads of the memory from the states considered.

Listing C.1: A four line modification to OpenSSL 1.0.1j, which introduces a client authentication bypass at a configurable depth (`BACKDOOR_DEPTH`). In the first part, we hijack a counter, the `read_sequence` counter, and conditionally increment it each time an unexpected data frame is read whilst waiting on the ClientCertificateVerify signature. In the second part, if this counter reaches the specified depth value, invalid ClientCertificateVerify signatures are ignored, and the handshake proceeds to the ChangeCipherSpec state. After this point, the `read_sequence` is automatically reset by the implementation for use in later MAC calculations.

```
@@ /ssl/s3_pkt.c:139 ssl3_read_n()
         SSL3_BUFFER *rb;
+++ // Backdoor pt 1: Set-up
+++ if(s->state == SSL3_ST_SR_CERT_VRFY_A)
+++     ++(*s->s3->read_sequence);
         if (n <= 0) return n;
         rb = &(s->s3->rbuf);
@@ /ssl/s3_srvr.c:683  ssl3_accept()
         ret=ssl3_get_cert_verify(s);
--- if (ret <= 0) goto end;
+++ // Backdoor pt 2: Activation
+++ if(*s->s3->read_sequence != BACKDOOR_DEPTH
+++     && ret <= 0) {
+++     goto end;
+++ } else {
      ...
      s->state=SSL3_ST_SR_FINISHED_A;
      s->init_num=0;
+++ }
```

Misleading OpenSSL Documentation

When testing the built-in server of OpenSSL, we found that configuring it to enforce client authentication is rather error-prone. In particular, after an initial reading of the documentation for how to do so, we came across the option below:

```
-verify int, -Verify int
    The verify depth to use. This specifies the
    maximum length of the client certificate
    chain and makes the server request a
    certificate from the client. With the
    -verify  option a certificate is requested
    but the client does not have to send one,
    with the -Verify option the client must
    supply a certificate or an error occurs.
```

Following this, we used the `-Verify` option when starting the built-in server. Surprisingly though, our learned models indicated that the server accepted invalid client certificates. At first glance, this looked like a severe vulnerability, however, upon closer inspection of the documentation we found an extra option was required:

```
-verify_return_error
    Verification errors normally just print a
    message but allow the connection to
    continue, for debugging purposes. If this
    option is used, then verification errors
    close the connection.
```

Our online investigation revealed we were not alone in our confusion. A recent Github issue[1] describes the same misleading configuration, and resulted in a patch to the OpenSSL client implementation[2]. Unfortunately, a similar patch was not written for the server implementation, resulting in a confusing disparity between the command-line options for the client and server. We have reported this issue to the OpenSSL developers.

[1]https://github.com/openssl/openssl/issues/8079
[2]https://github.com/openssl/openssl/pull/8080

Bibliography

[1] Aarts, F., De Ruiter, J., and Poll, E. Formal models of bank cards for free. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (2013), IEEE, pp. 461–468.

[2] Aarts, F., Fiterau-Brostean, P., Kuppens, H., and Vaandrager, F. Learning register automata with fresh value generation. In *International Colloquium on Theoretical Aspects of Computing* (2015), Springer, pp. 165–183.

[3] Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., and Vaandrager, F. Automata learning through counterexample guided abstraction refinement. In *International Symposium on Formal Methods* (2012), Springer, pp. 10–27.

[4] Aarts, F., Jonsson, B., and Uijen, J. Generating models of infinite-state communication protocols using regular inference with abstraction. In *IFIP International Conference on Testing Software and Systems* (2010), Springer, pp. 188–204.

[5] Aarts, F., Kuppens, H., Tretmans, J., Vaandrager, F., and Verwer, S. Learning and testing the bounded retransmission protocol. In *International Conference on Grammatical Inference* (2012), pp. 4–18.

[6] Aarts, F., Schmaltz, J., and Vaandrager, F. Inference and abstraction of the biometric passport. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (2010), Springer, pp. 673–686.

[7] Aarts, F., and Vaandrager, F. Learning I/O Automata. In *International Conference on Concurrency Theory* (2010), Springer, pp. 71–85.

[8] Aichernig, B. K., Bloem, R., Ebrahimi, M., Tappler, M., and Winter, J. Automata learning for symbolic execution. In *2018 Formal Methods in Computer Aided Design (FMCAD)* (2018), IEEE, pp. 1–9.

[9] Alur, R., and Dill, D. L. A theory of timed automata. *Theoretical computer science 126*, 2 (1994), 183–235.

[10] Amini, P., and Portnoy, A. Sulley: Pure Python fully automated and unattended fuzzing framework. *May* (2013).

[11] An, J., Chen, M., Zhan, B., Zhan, N., and Zhang, M. Learning one-clock timed automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2020), Springer, pp. 444–462.

[12] Angluin, D. Learning regular sets from queries and counterexamples. *Information and computation 75*, 2 (1987), 87–106.

[13] Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., and Vigna, G. Snooze: toward a stateful network protocol fuzzer. In *International Conference on Information Security* (2006), Springer, pp. 343–358.

[14] Bauer, K., Gonzales, H., and McCoy, D. Mitigating evil twin attacks in 802.11. In *Performance, computing and communications conference, 2008. IPCCC 2008. IEEE International* (2008), IEEE, pp. 513–516.

[15] Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. Scalable, behavior-based malware clustering. In *NDSS* (2009), vol. 9, Citeseer, pp. 8–11.

[16] Bayer, U., Moser, A., Kruegel, C., and Kirda, E. Dynamic analysis of malicious code. *Journal in Computer Virology 2*, 1 (2006), 67–77.

[17] Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y., and Zinzindohoue, J. K. A messy state of the union: Taming the composite state machines of TLS. In *Security and Privacy (SP), 2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 535–552.

[18] Beurdouche, B., Delignat-Lavaud, A., Kobeissi, N., Pironti, A., and Bhargavan, K. FLEXTLS: A Tool for Testing TLS Implementations. In *9th USENIX Workshop on Offensive Technologies (WOOT) 15)* (2015).

[19] Bhargavan, K., Lavaud, A. D., Fournet, C., Pironti, A., and Strub, P. Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 98–113.

[20] Bosman, E. ptrace-burrito. Retrieved 3 September 2020 from https://github.com/brainsmoke/ptrace-burrito, 2020.

[21] Bossert, G., and Guihery, F. Security evaluation of communication protocols in common criteria. In *Proceedings of the IEEE International Conference on Communications, Ottowa, ON, Canada* (2012), pp. 10–15.

[22] Bossert, G., Guihéry, F., Hiet, G., et al. Netzob: un outil pour la rétroconception de protocoles de communication. *SSTIC 2012* (2012), 43.

[23] Bremond, N., and Groz, R. Case studies in learning models and testing without reset. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2019), IEEE, pp. 40–45.

[24] Brenza, S., Pawlowski, A., and Pöpper, C. A practical investigation of identity theft vulnerabilities in eduroam. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2015), ACM, p. 14.

[25] Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A. Model-based testing of reactive systems. In *Volume 3472 of Springer LNCS* (2005), Springer.

[26] Brubaker, C., Jana, S., Ray, B., Khurshid, S., and Shmatikov, V. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 114–129.

[27] Buildroot Association. Buildroot. Retrieved 3 September 2020 from https://buildroot.org/, 2020.

[28] Butti, L., and Tinnes, J. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology 4*, 1 (2008), 25–37.

[29] Cadar, C., Dunbar, D., Engler, D. R., et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[30] Cassel, S., Howar, F., Jonsson, B., Merten, M., and Steffen, B. A succinct canonical register automaton model. In *International Symposium on Automated Technology for Verification and Analysis* (2011), Springer, pp. 366–380.

[31] Chen, P., and Chen, H. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 711–725.

[32] Chen, Y., Lan, T., and Venkataramani, G. Exploring effective fuzzing strategies to analyze communication protocols. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation* (2019), pp. 17–23.

[33] Chen, Y., Song, L., Xing, X., Xu, F., and Wu, W. Automated finite state machine extraction. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation* (2019), pp. 9–15.

[34] Cho, C. Y., Babic, D., Poosankam, P., Chen, K. Z., Wu, E. X., and Song, D. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security Symposium* (2011), vol. 139.

[35] Choi, W., Necula, G., and Sen, K. Guided GUI testing of Android apps with minimal restart and approximate learning. *Acm Sigplan Notices 48*, 10 (2013), 623–640.

[36] Chothia, T., de Ruiter, J., and Smyth, B. Modeling and analysis of a hierarchy of distance bounding attacks. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association.

[37] Chothia, T., Garcia, F. D., Heppel, C., and McMahon Stone, C. Why Banker Bob (still) Can't Get TLS Right: A Security Analysis of TLS in Leading UK Banking Apps. In *Financial Cryptography and Data Security* (2017), Springer.

[38] Chow, T. S. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, 3 (1978), 178–187.

[39] CodeApe123. Hostapd porting and use. Retrieved 9 January 2020 from https://blog.csdn.net/sean_8180/article/details/86496922, 2020.

[40] Comparetti, P. M., Wondracek, G., Kruegel, C., and Kirda, E. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 110–125.

[41] Crisan, C. motioneyeos. Retrieved 3 September 2020 from https://github.com/ccrisan/motioneyeos, 2020.

[42] Daniel, L.-A., Poll, E., and de Ruiter, J. Inferring OpenVPN state machines using protocol state fuzzing. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (2018), IEEE, pp. 11–19.

[43] de Moura, L. M., and Bjørner, N. Z3: an efficient SMT solver. In *Proceedings of the 14th Tools and Algorithms for the Construction and Analysis of Systems Conference* (2008).

[44] de Ruiter, J., and Poll, E. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)* (Aug. 2015), USENIX Association.

[45] Duchene, J., Le Guernic, C., Alata, E., Nicomette, V., and Kaâniche, M. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques 14*, 1 (2018), 53–68.

[46] Durumeric, Z., Adrian, D., Mirian, A., Bailey, M., and Halderman, J. A. A search engine backed by internet-wide scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 542–553.

[47] Eastlake, D., et al. Transport layer security (TLS) extensions: Extension definitions.

[48] Eddington, M. Peach fuzzing platform. *Peach Fuzzer 34* (2011).

[49] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), CCS '12.

[50] Feng, L., Lundmark, S., Meinke, K., Niu, F., Sindhu, M. A., and Wong, P. Y. Case studies in learning-based testing. In *IFIP International Conference on Testing Software and Systems* (2013), Springer, pp. 164–179.

[51] Fisher, D. Final Report on DigiNotar Hack Shows Total Compromise of CA Servers. *Retrieved September 8* (2012), 2013.

[52] Fiterău-Broştean, P., Janssen, R., and Vaandrager, F. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification* (2016), Springer, pp. 454–471.

[53] Fiterau-Brostean, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., and Somorovsky, J. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 2523–2540.

[54] Fiterău-Broştean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F., and Verleg, P. Model Learning and Model Checking of SSH Implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software* (2017), SPIN 2017, ACM, pp. 142–151.

[55] Fluhrer, S., Mantin, I., Shamir, A., et al. Weaknesses in the key scheduling algorithm of RC4. In *Selected areas in cryptography* (2001), vol. 2259, Springer, pp. 1–24.

[56] Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., and Rieck, K. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems* (2015), Springer, pp. 330–347.

[57] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), CCS '12, ACM, pp. 38–49.

[58] Gonzales, H., Bauer, K., Lindqvist, J., McCoy, D., and Sicker, D. Practical defenses for evil twin attacks in 802.11. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE* (2010), IEEE, pp. 1–6.

[59] Grinchtein, O., Jonsson, B., and Leucker, M. Learning of event-recording automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 379–395.

[60] Group, I. . W., et al. IEEE standard for information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications. *IEEE Std 802*, 11 (2010).

[61] Groz, R., Bremond, N., and Simao, A. Using adaptive sequences for learning non-resettable FSMS. In *International Conference on Grammatical Inference* (2019), pp. 30–43.

[62] He, C., and Mitchell, J. C. Analysis of the 802.11i 4-way handshake. In *Proceedings of the 3rd ACM workshop on Wireless security* (2004), ACM, pp. 43–50.

[63] He, C., Sundararajan, M., Datta, A., Derek, A., and Mitchell, J. C. A modular correctness proof of IEEE 802.11i and TLS. In *Proceedings of the 12th ACM conference on Computer and communications security* (2005), ACM, pp. 2–15.

[64] Hercog, D. Selective-repeat protocol with multiple retransmit timers and individual acknowledgments. *Elektrotehniski Vestnik 82*, 1/2 (2015), 55.

[65] Hex-Rays. IDA Pro. Retrieved 3 September 2020 from https://www.hex-rays.com/products/ida/, 2020.

[66] Irfan, M. N., Groz, R., and Oriat, C. Optimising Angluin algorithm L* by minimising the number of membership queries to process counterexamples. In *Zulu Workshop, Valencia* (2010), p. 134.

[67] Irfan, M. N., Oriat, C., and Groz, R. Angluin style finite state machine inference with non-optimal counterexamples. In *Proceedings of the First International Workshop on Model Inference In Testing* (2010), ACM, pp. 11–19.

[68] Isberner, M., Howar, F., and Steffen, B. The TTT algorithm: a redundancy-free approach to active automata learning. In *International Conference on Runtime Verification* (2014), Springer, pp. 307–322.

[69] Isberner, M., Howar, F., and Steffen, B. The open-source learnlib. In *International Conference on Computer Aided Verification* (2015), Springer, pp. 487–495.

[70] Janssen, M. Combining learning with fuzzing for software deobfuscation.

[71] Jonsson, B. Modular verification of asynchronous networks. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing* (1987), pp. 152–166.

[72] Jonsson, B., and Vaandrager, F. Learning Mealy machines with timers. Retrieved 3 September 2020 from https://www.sws.cs.ru.nl/publications/papers/fvaan/MMT/full.pdf, 2018.

[73] Kaminski, M., and Francez, N. Finite-memory automata. *Theoretical Computer Science 134*, 2 (1994), 329–363.

[74] Krueger, T., Gascon, H., Krämer, N., and Rieck, K. Learning stateful models for network honeypots. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence* (2012), pp. 37–48.

[75] Krueger, T., Krämer, N., and Rieck, K. Asap: Automatic semantics-aware analysis of network payloads. In *International Workshop on Privacy and Security Issues in Data Mining and Machine Learning* (2010), Springer, pp. 50–63.

[76] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., et al. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 183–196.

[77] Lynch, N. A., and Tuttle, M. R. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing* (1987), ACM, pp. 137–151.

[78] Maler, O., and Pnueli, A. On the learnability of infinitary regular sets. *Information and Computation 118*, 2 (1995), 316–326.

[79] Margaria, T., Niese, O., Raffelt, H., and Steffen, B. Efficient test-based model generation for legacy reactive systems. In *High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International* (2004), IEEE, pp. 95–100.

[80] McMahon Stone, C. Wifi learner. https://chrismcmstone.github.io/wifi-learner/, 2018.

[81] McMahon Stone, C., Chothia, T., and de Ruiter, J. Extending automated protocol state learning for the 802.11 4-way handshake. In *European Symposium on Research in Computer Security* (2018), Springer, pp. 325–345.

[82] McMahon Stone, C., Chothia, T., and Garcia, F. D. Spinner: Semi-Automatic Detection of Pinning Without Hostname Verification. In *Proceedings of the 33nd Annual Conference on Computer Security Applications* (2017), ACM.

[83] McMahon Stone, C., Thomas, S. L., Vanhoef, M., and Chothia, T. Grey-box Protocol State Machine Learning. In *Under Submission*.

[84] Mendonça, M., and Neves, N. Fuzzing Wi-Fi drivers to locate security vulnerabilities. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European* (2008), IEEE, pp. 110–119.

[85] Mitchell, C. Security analysis and improvements for ieee 802.11 i. In *The 12th Annual Network and Distributed System Security Symposium (NDSS'05) Stanford University, Stanford* (2005), pp. 90–110.

[86] Moixe, M. New Tricks For Defeating SSL in Practice. In *BlackHat Conference, USA* (2009).

[87] Möller, B., Duong, T., and Kotowicz, K. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.

[88] Moon, S.-J., Helt, J., Yuan, Y., Bieri, Y., Banerjee, S., Sekar, V., Wu, W., Yannakakis, M., and Zhang, Y. Alembic: Automated model inference for stateful network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 699–718.

[89] Niese, O. *An integrated approach to testing complex systems.* PhD thesis, Universität Dortmund, 2003.

[90] Oltrogge, M., Acar, Y., Dechand, S., Smith, M., and Fahl, S. To Pin or Not to Pin-Helping App Developers Bullet Proof Their TLS Connections. In *USENIX Security* (2015), pp. 239–254.

[91] Oltrogge, M., Huaman, N., Amft, S., Acar, Y., Backes, M., and Fahl, S. Why Eve and Mallory Still Love Android: Revisiting TLS (In) Security in Android Applications. In *30th USENIX Security Symposium (USENIX Security 21).*

[92] Pereyda, J. boofuzz: Network protocol fuzzing for humans. *Accessed: Feb 17* (2017).

[93] Perl, H., Fahl, S., and Smith, M. You won't be needing these any more: On removing unused certificates from trust stores. In *International Conference on Financial Cryptography and Data Security* (2014), Springer, pp. 307–315.

[94] Pferscher, A., Aichernig, B., and Tappler, M. From passive to active: Learning timed automata efficiently. In *12th NASA Formal Methods Symposium* (2020).

[95] Pham, V.-T., Böhme, M., and Roychoudhury, A. Aflnet: A greybox fuzzer for network protocols. In *Proc. IEEE International Conference on Software Testing, Verification and Validation (Testing Tools Track)* (2020).

[96] Philippe Arteau. Weaknesses in Java TLS Host Verification. Retrieved 30 Octover 2020 from https://www.gosecure.net/blog/2020/10/27/weakness-in-java-tls-host-verification/, 2020.

[97] Poeplau, S., and Francillon, A. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20), pages=181–198, year=2020*.

[98] QUARKSLAB. Triton. Retrieved 3 September 2020 from https://triton.quarkslab.com/, 2020.

[99] Raffelt, H., Steffen, B., Berg, T., and Margaria, T. Learnlib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer (STTT) 11*, 5 (2009), 393–407.

[100] Rasool, A., Alpár, G., and de Ruiter, J. State machine inference of QUIC. *arXiv preprint arXiv:1903.04384* (2019).

[101] Reaves, B., Scaife, N., Bates, A., Traynor, P., and Butler, K. R. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world. In *24th USENIX Security Symposium (USENIX Security 15)* (2015).

[102] Riley, M. Nsa said to have used heartbleed bug, exposing consumers. *Bloomberg.com, Available: https://www.bloomberg.com/news/articles/2014-04-11/nsa-said-to-have-used-heartbleed-bug-exposing-consumers* (2014).

[103] Rivest, R. L., and Schapire, R. E. Inference of finite automata using homing sequences. *Information and Computation 103*, 2 (1993), 299–347.

[104] Sen, K., Marinov, D., and Agha, G. Cute: a concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes 30*, 5 (2005), 263–272.

[105] Shahbaz, M., and Groz, R. Inferring Mealy Machines. *FM 9* (2009), 207–222.

[106] Shahbaz, M., and Groz, R. Analysis and testing of black-box component-based systems by inferring partial models. *Software Testing, Verification and Reliability 24*, 4 (2014), 253–288.

[107] Sivakorn, S., Argyros, G., Pei, K., Keromytis, A. D., and Suman, J. HVLearn: Automated Black-box Analysis of Hostname Verification in SSL/TLS Implementations. In *Security and Privacy (SP), 2017 IEEE Symposium on* (2017), IEEE.

[108] Sleevi, R. Intent to deprecate and remove: Trust in existing symantec-issued certificates. *Chromium development forum* (2017).

[109] Slowinska, A., Stancescu, T., and Bos, H. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS* (2011).

[110] Somorovsky, J. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 1492–1504.

[111] Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., and Khan, L. SMV-Hunter: Large scale, automated detection of SSL/TLS Man-in-the-Middle vulner-

abilities in Android apps. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)* (2014), Citeseer.

[112] Steinberg, J. Massive internet security vulnerability–here's what you need to do. *Forbes. Available: http://www.forbes.com/sites/josephsteinberg/2014/04/10/massive-internet-securityvulnerability-you-are-at-risk-what-you-need-to-do* (2014).

[113] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices 39*, 11 (2004), 85–96.

[114] Tappler, M., Aichernig, B. K., and Bloem, R. Model-based testing iot communication via active automata learning. In *2017 IEEE International conference on software testing, verification and validation (ICST)* (2017), IEEE, pp. 276–287.

[115] Tews, E., and Beck, M. Practical attacks against WEP and WPA. In *Proceedings of the second ACM conference on Wireless network security* (2009), ACM, pp. 79–86.

[116] Vanhoef, M. Krack attack scripts. Retrieved 30 January 2020 from https://github.com/vanhoefm/krackattacks-scripts, 2021.

[117] Vanhoef, M., and Piessens, F. Predicting, Decrypting, and Abusing WPA2/802.11 Group Keys. In *USENIX Security Symposium* (2016), pp. 673–688.

[118] Vanhoef, M., and Piessens, F. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 24th ACM Conference on Computer and Communication Security* (2017), ACM.

[119] Vanhoef, M., Schepers, D., and Piessens, F. Discovering logical vulnerabilities in the wi-fi handshake using model-based testing. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 360–371.

[120] Verwer, S., de Weerdt, M., and Witteveen, C. The efficiency of identifying timed automata and the power of clocks. *Information and Computation 209*, 3 (2011), 606–625.

[121] Wagner, D., Schneier, B., et al. Analysis of the ssl 3.0 protocol. In *The Second USENIX Workshop on Electronic Commerce Proceedings* (1996), vol. 1, pp. 29–40.

[122] Wang, L., and Srinivasan, B. Analysis and improvements over DoS attacks against IEEE 802.11i standard. In *Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on* (2010), vol. 2, IEEE, pp. 109–113.

[123] Wang, T., Wei, T., Lin, Z., and Zou, W. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS* (2009), Citeseer.

[124] Wang, Y., Zhang, Z., Yao, D. D., Qu, B., and Guo, L. Inferring protocol state machine from network traces: a probabilistic approach. In *International Conference on Applied Cryptography and Network Security* (2011), Springer, pp. 1–18.

[125] Wondracek, G., Comparetti, P. M., Kruegel, C., Kirda, E., and Anna, S. S. S. Automatic network protocol analysis. In *NDSS* (2008), vol. 8, pp. 1–14.

[126] Zalewski, M. American fuzzy lop, 2014.