SYSTEMATIC EVALUATIONS OF SECURITY MECHANISM DEPLOYMENTS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Sze Yiu Chau

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

Dedicated to my wife Candise, who stood by me through thick and thin, and
to my loving parents, whose sacrifices and selfless support made everything possible.

ACKNOWLEDGMENTS

My most sincere gratitude goes to my advisors Dr. Ninghui Li and Dr. Aniket Kate, for the support and guidance that I have received from them. It is my absolute honor and pleasure to be given the chance to work with such inspiring and productive mentors, and they have been tremendously helpful in my quest of becoming an independent researcher. This dissertation would not have been possible without their insight and motivation.

I would also like to thank the committee members of my preliminary and final exams, Dr. Dongyan Xu, Dr. Elisa Bertino, and Dr. Lin Tan, for their constructive comments and insightful suggestions, which were very helpful in improving the overall quality of this dissertation.

Heartfelt thanks must be given to Dr. Omar Chowdhury, who not only served in my committee as an external member, but also gave me an enormous amount of encouragement and advice throughout my journey at Purdue.

I have sincere appreciation for each of my lab mates, including Pedro Moreno-Sanchez, Duc Le, Mohsen Minaei, Easwar Mangipudi, Debajyoti Das, Donghang Lu, Adithya Bhat, Syed Rafiul Hussain, Endadul Hoque, Weining Yang, Haining Chen, Dong Su, Huangyi Ge, Weicheng Wang, Tianhao Wang, Wuwei Zhang, Zitao Li, and Jiangcheng Li. I am very fortunate to have met them, for their friendship and the countless discussions that we had made my Purdue years much more fun and interesting. I also had the pleasure of collaborating with some other young and brilliant minds, including Moosa Yahyazadeh, Joyanta Debnath, Bincheng Wang, Jianxiong Wang and Durga Keerthi Mandarapu.

Thanks should also be given to Dr. Grace Ngai and Dr. Rocky Chang, for their advice and continuous support made me dare to dream, as well as my numerous friends from The Hong Kong Polytechnic University and Diocesan Boys' School, who had been very kind and caring.

Finally, my deepest gratitude goes to my parents and my wife, for the perpetual love, patience and support that I have received from them throughout my life.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Chau, Sze Yiu Ph.D., Purdue University, August 2019. Systematic Evaluations of Security Mechanism Deployments. Major Professors: Ninghui Li and Aniket Kate.

In a potentially hostile networked environment, a large diversity of security mechanisms with varying degree of sophistication are being deployed to protect valuable computer systems and digital assets. While many competing implementations of similar security mechanisms are available in the current software development landscape, the robustness and reliability of such implementations are often overlooked, resulting in exploitable flaws in system deployments. In this dissertation, we systematically evaluate implementations of security mechanisms that are deployed in the wild. First, we examine how content distribution applications on the Android platform control access to their multimedia contents. With respect to a well-defined hierarchy of adversarial capabilities and attack surfaces, we find that many content distribution applications, including that of some world-renowned publications and streaming services, are vulnerable to content extraction due to the use of unjustified assumptions in their security mechanism designs and implementations. Second, we investigate the validation logic of X.509 certificate chains as implemented in various open-source TLS libraries. X.509 certificates are widely used in TLS as a means to achieve authentication. A validation logic that is overly restrictive could lead to the loss of legitimate services, while an overly permissive implementation could open door to impersonation attacks. Instead of manual analysis and unguided fuzzing, we propose a principled approach that leverages symbolic execution to achieve better coverage and uncover logical flaws that are buried deep in the code. We find that many TLS libraries deviate from the specification. Finally, we study the verification of RSA signatures, as specified in the PKCS#1 v1.5 standard, which is widely used in many security-critical network protocols. We propose an approach to automatically generate meaningful concolic test cases for this particular problem, and design and implement a provenance tracking mechanism to assist root-cause analysis in general. Our investigation revealed that several crypto and IPSec implementations are susceptible to new variants of the Bleichenbacher low-exponent signature forgery.

1. INTRODUCTION

In a world full of potential adversaries, networked computer systems rely on various security mechanisms to protect their availability as well as to guarantee communication confidentiality and integrity. Businesses in the digital age depends on reliable and trustworthy systems to thrive. End user security and privacy could be in serious jeopardy without adequate protections. It is hence important to not only have well designed security mechanisms but also robust and reliable implementations that faithfully fulfill the intended security goals when they are deployed.

In this dissertation, we systematically evaluate implementations of widely deployed security mechanisms. We attempt to make contributions in two directions: 1) identifying new implementation weaknesses that are potentially exploitable and help the development community fix those and avoid future pitfalls; 2) advancing the state of the art in analyzing semantic correctness of security-critical protocol implementations. Taking a ***top-down*** perspective, we begin with a study on content distribution applications on Android, where various access control mechanisms are deployed at different layers to protect multimedia contents, the main assets in their business models. We then set our focus on TLS connections, the *de facto* standard for encrypting Internet traffic. Specifically, we investigate the logic of X.509 certificate chain validation as implemented in various open-source TLS libraries, with a principled symbolic analysis approach. Finally we investigate the verification of RSA signatures, with a more automated approach and new improvements on root-cause analysis.

1.1 Content Distribution Applications on Android

Mobile devices are becoming the default platform for multimedia content consumption. Such a thriving business ecosystem has drawn interests from content distributors to develop applications that can reach a large number of audiences. The business-edge of content delivery applications crucially relies on being able to effectively arbitrate the purchase and delivery of contents, and govern the access of contents with respect to usage control policies, on a plethora of consumer devices. Content protection on mobile platforms, especially in the absence of Trusted Execution Environment (TEE), is a challenging endeavor where developers often have to resort to ad-hoc deterrence-based defenses. In Chapter 3 we systematically evaluate the effectiveness of content protection mechanisms embraced by vendors of content delivery applications, with respect to a hierarchy of adversaries with varying degrees of real-world capabilities. Our analysis of 141 vulnerable applications uncovered that, in many cases, due to developers' unjustified trust assumptions about the underlying platforms and technologies, adversaries can obtain unauthorized and unrestricted access to contents offered by the applications, sometimes without even needing to reverse engineer the deterrence-based defenses. Some weaknesses in the applications can also severely impact users' security and privacy. All our findings have been responsibly disclosed to the corresponding application vendors.

1.2 X.509 Certificate Chain Validation

The X.509 Public-Key Infrastructure has long been used in the TLS protocol to achieve authentication. A recent trend of Internet-of-Things (IoT) systems employing small footprint TLS libraries for secure communication has further propelled its prominence. The security guarantees provided by X.509 hinge on the assumption that the underlying implementation rigorously scrutinizes X.509 certificate chains, and accepts only the valid ones. Noncompliant implementations of X.509 can potentially lead to attacks and/or interoperability issues. In the literature, black-box fuzzing has been used to find flaws in X.509 validation implementations [1,2]. While black-box fuzzing makes a good attempt in revealing the existence of implementation problems, especially when source code is not available, there are limitations of such approach: 1) given a particular test case that indicates an error, it is often not easy to account for the exact root causes; 2) it lacks guarantees on coverage of the code being tested, especially in the unguided setting; 3) each randomly generated test case could contain multiple problems that might mask each other, making results difficult to interpret. To thoroughly analyze X.509 implementations in small footprint TLS libraries, we take the complementary approach of using *symbolic execution*. Our work attempts to take advantage of the fact that when the underlying source code is available, one can infer useful information out of the code, and perform testing using such information to achieve better code coverage.

While symbolic execution is a technique proven to be effective in finding software implementation flaws, it can also be leveraged to expose noncompliance in X.509 implementations. Directly applying an off-the-shelf symbolic execution engine on TLS libraries is, however, not practical due to the problem of path explosion. In Chapter 4, we propose the use of *SymCerts*, which are X.509 certificate chains carefully constructed with a mixture of symbolic and concrete values. Utilizing SymCerts and some domain-specific optimizations, one can symbolically execute the certificate chain validation code of each library and extract path constraints describing its accepting and rejecting certificate universes. These path constraints help us to easily identify missing checks in different libraries. For exposing subtle but intricate noncompliance with the X.509 standard, we cross-validate the constraints extracted from different libraries to find further implementation flaws. Our analysis of 9 small footprint X.509 implementations has uncovered 48 instances of noncompliance. Many findings and suggestions provided by us have already been incorporated by the vendors in newer versions of their libraries.

1.3 PKCS#1 v1.5 RSA Signature Verification

While known implementation flaws in cryptographic libraries can sometimes be abstracted into certain patterns to enable the measurement of scale and spread of the vulnerabilities [3–6], current research efforts on finding attacks against cryptographic implementations often rely on *manual analysis of the code* with respect to the standard specification, and then design mathematical exploitations of the identified flaws [3, 7, 8].

In Chapter 5 we discuss the possibility of automating the identification of implementation flaws in cryptographic glue protocols. Because of the restrictive assumptions used in designing cryptographic constructs, in reality, additional glue protocols are often needed to generalize such constructs into being able to handle inputs of diverse length and formats. Sometimes glue protocols are also used to wrap around cryptographic constructs for exploiting the duality of certain security guarantees to achieve alternative properties. Our overarching goal is to develop a systematic approach for analyzing the semantic correctness of implementations of such glue protocols that are deployed in practice, and enabling cryptographers to only concentrate on devising mathematical exploitations.

As a case study, we systematically analyze implementations of RSA signature verification, the robustness and reliability of which is crucial for achieving authentication and integrity guarantees. Interestingly, most previous work on analyzing certificate chain validation neglect to investigate the implementation of signature verification [1, 2, 9, 10]. Faulty signature verifiers are known to enable attackers forging digital signatures without the possession of the private key [7, 8, 11–15].

At the time of writing, RSA remains one of the most widely-used asymmetric cryptosystem. The PKCS#1 standard in particular defines several versions of encryption, decryption and signature schemes based on the RSA algorithm. Despite the existence of newer schemes with provable security like RSA-PSS introduced in the version 2.0 specification [RFC8017], the version 1.5 standard continues to be extensively used in the Web PKI and other security-critical network protocols like *SSH* [RFC4253] and *IKEv2* [RFC7296]. As we will explain later in Chapter 5, to our surprise, even after a decade since the discovery of the original vulnerability [8], implementations still fail to faithfully and robustly implement the prescribed PKCS#1 v1.5 glue protocol, resulting in new variants of attacks.

The diverse glue components involved in PKCS#1 v1.5 makes it a good candidate for demonstrating the effectiveness of our approach in analyzing semantic correctness. In contrast to X.509 certificates, PKCS#1 v1.5 signatures consist of components of variable lengths (*e.g.* padding and other metadata) that complement each other, hence the technique of choosing fixed lengths as done in previous work [9] could potentially miss out on numerous meaningful test cases. For achieving better coverage and a higher degree of automation, we propose to use a technique dubbed "meta-level search", where symbolic variables are not only used as test inputs, but also indicate how components of inputs can be changed in lengths and combined together, by exploiting the linear relations that exist among the various components. This enables the automatic generation of many meaningful concolic test inputs, an improvement over manually constructing concolic inputs as done in previous work [9].

To facilitate root-cause analysis of implementation flaws identified with our approach, we design and develop a *constraint provenance tracking* (CPT) mechanism that maps the different clauses of each path constraint generated by symbolic execution to their source level origin, which can be used to understand where certain decisions were being made inside the source tree.

Most of the flaws in PKCS#1 v1.5 signature verification found in this research have already been reported to and fixed by maintainers of the corresponding software.

1.4 Thesis Statement

This thesis focuses on demonstrating the following statement:

Current security mechanisms that are widely deployed still contain hidden but exploitable weaknesses, and with sufficient domain knowledge, such weaknesses can be found in a systematic manner.

1.5 Contributions

The technical contributions of this thesis can be broadly partitioned into two categories.

i. Identifying weaknesses and avoiding future pitfalls

Our systematic evaluations of content distribution applications, as well as implementations of X.509 certificate validation and PKCS#1 v1.5 signature verification, have uncovered numerous weaknesses that are exploitable. By documenting and dissecting the root-causes of said weaknesses, we assist future development of security mechanisms with the weakness patterns identified, so that similar design and implementation pitfalls can be avoided.

ii. Advancing the state of the art in semantic correctness analysis

We advance the state of the art in analyzing the semantic correctness of protocol implementations. While black-box fuzzing has been the dominant approach in finding software implementation flaws, especially low-level memory safety issues, our position is that for the analysis of semantic correctness (*i.e.*, whether an implementation faithfully adhere to the protocol specification), symbolic analysis can often provide better code coverage and a more useful formula-based abstraction of the implemented logic. Semantic correctness is particularly interesting to analyze because memory errors can be easily avoided by using memory-safe programming languages, but the same is not true for logical flaws. We also demonstrate how to leverage domain knowledge to make symbolic execution practical for specific problems.

1.6 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides an overview of previous literature related to the discovery and exploitation of flaws in security mechanism deployments. Chapter 3 presents our systematic evaluation of content distribution applications on the Android platform. Chapter 4 discusses the problem of X.509 certificate validation, and how symbolic execution can be applied to analyze the implemented validation logic in open-source TLS libraries. Chapter 5 gives a discussion on how to symbolically analyze implementations of cryptographic glue protocols and make root-cause analysis easier, using PKCS#1 v1.5 RSA signatures as the main case study example. Chapter 6 concludes the dissertation.

2. RELATED WORK

In this chapter, we briefly review related work on attacks against design and implementation flaws in deployed security mechanisms, measurements of deployment issues, as well as software testing techniques and their application on evaluating security-critical protocol implementations.

2.1 Measuring the spread and scale of deployment issues

A prominent line of research is measurement studies on issues facing deployments of security mechanisms, for example, environmental threats and known vulnerabilities. This helps one to understand and evaluate the current deployment practices.

Vulnerable keys and other cryptographic weaknesses. Zhang et al. [6] studied how X.509 certificates were reissued and/or revoked after the discovery of the OpenSSL Heartbleed vulnerability. Heninger et al. [4] studied the spread of weak RSA and DSA keys at the Internet scale by scanning X.509 certificates used in TLS connections and SSH host keys. They showed that the majority of vulnerable RSA and DSA keys are due to the use of some insecure random number generators. Hastings et al. [5] presented a follow-up analyze on the measures taken by vendors and end users regarding the advisory on weak RSA keys. According to their findings, end users seem reluctant in patching their vulnerable software, and some vendors did not release any patches at all. Valenta et al. [3] surveyed the presence of known vulnerabilities against elliptic curve implementations by performing internet-wide scans. A recent study showed that a considerable number of servers on the Internet are still vulnerable to Bleichenbacher's padding oracle attack [16].

Forged X.509 certificates. Huang et al. [17] designed a client-side applet to monitor and report X.509 certificate chains that were actually presented to clients. Their study discovered about 6 thousand forged certificates in over 3 million connections, and showed that not just malware but surveillance devices as well as anti-virus software are also forging certificates to tamper with SSL/TLS connections.

TLS interception. The security guarantees provided by TLS can indeed be affected by even benign software and networking equipments. Studies found that many anti-virus and parental control software [18, 19] as well as enterprise-grade network appliances [19, 20] attempt to intercept TLS connections for various reasons. It was shown that many interceptors fail to properly validate X.509 certificates and might be willing to offer and accept weak ciphersuites, hence significantly degrade the security of the TLS connections being intercepted [18–20].

2.2 Achieving more robust implementations and deployments

The research community has seen numerous efforts on how to better achieve security, through the means of refactoring specifications, proposing good development practices, enforcing correct library API usages and formally verifying implementations. Here we give a brief account of such efforts.

TLS state machine and high-confidence implementations. Attempts were made on building high-confidence TLS implementations with a focus on correct state transitions and cryptographic primitives, using re-engineered protocol specification and modular code base [21], as well as verified code along with security proofs [22]. Beurdouche et al. [23] designed a tool that uses a verified implementation as a reference to test the state machine of other SSL/TLS implementations. At the time of writing, existing work on reference SSL/TLS implementations do not include a formally verified X.509 certificate validation logic.

Incorrect and insecure usage of TLS library APIs. Another direction of research regarding deployments of TLS is whether applications are developed to make correct usage of the API of a given TLS library. Georgiev et al. [24] crafted a handful of attack certificates to attempt MITM attacks against various SSL/TLS library-using applications, and showed that application developers often misunderstand and misuse APIs, resulting in vulnerabilities. Further discussions on false beliefs of developers, exploits on TLS-using applications and correct usage of TLS can be found in [25]. He et al. [26] showed how to use static analysis to vet and identify vulnerable API usage in applications. Yun et al. [27] proposed a fully automated system called APISAN that can infer correct API usage from other some sample references, and use the inferred information to find inconsistent API usages in other applications.

We note that this line of research is orthogonal to and complements our work presented in Chapters 4 to 5, as we are focused on how the underlying libraries providing those APIs are implementing the validation of X.509 certificates and PKCS#1 v1.5 RSA signatures. Problems in the library implementations would affect applications even if the application developers made no mistakes in using the APIs.

Protecting flawed certificate validation implementations. Since implementing a robust certificate validation logic is non-trivial and error-prone [1, 9], patching vulnerable implementations in a timely manner is another important aspect of actual deployments. Bates et al. [28] proposed to use dynamically linked objects and binary instrumentation to implement a defense layer, so that vulnerabilities in certificate validation implementations can be patched promptly, and insecure configurations can be overridden and proper extension handling can be enforced.

Mobile application weakness analysis. Given that mobile computing devices are becoming prevalent and ubiquitous, the security of applications tailor-made for such emergent platforms warrants specific analyses. For example, Reaves et al. [29] have carried out an analysis of 7 branchless mobile banking applications, and uncovered weak design and implementation practices including inadequate authentication and authorization checks, weak (or, non-standard) cryptographic primitive usage, predictable key usage, and sensitive information leakage. Other studies have shown that many back-end servers used to support the services of mobile applications have vulnerable authorization mechanisms [30], as well as insufficient request message checks that can lead to access token hijacks and password brute-force attacks [31].

Mobile application security standard. OWASP recently released version 1 of its Mobile Application Security Verification Standard [32], which attempts to standardize security requirements and verification levels that fit different application and threat scenario. Interestingly, for Intellectual Property protection, it recommends verification level `L1+R`. While `R` requires resiliency against reverse engineering, `L1` does not require key/certificate pinning, which as we will discuss in Sections 3.3.3 and 3.3.4, allows for relatively easy TLS interception and potential content protection bypass.

Implementing content protection. Some have suggested that the large file size of high-definition multimedia contents can be considered as a natural Digital Rights Management (DRM) mechanism [33], for which we disagree with. While large file size slightly hinder Internet sharing, DRM has a variety of other objectives like copy control, license expiration check and authorization that are beyond the scope of Internet sharing, especially in the era that the subscription-based streaming business is dominant.

2.3 Techniques for testing security-critical protocol implementations

Symbolic execution. Symbolic execution has been shown to be effective for finding low-level errors (*e.g.*, null dereferencing, buffer overrun, division by zero *etc.*) [34–43]. It has also been used for checking the equivalence of C functions [44, 45], for checking server–client interoperability of network protocols based on the set of packets accepted by them [46], for checking controllers in software-defined networks [47, 48], and for cross-checking different file system implementations to find semantic bugs [49]. Symbolic execution has also been used to assist the verification of network functions [50] and cryptographic implementations as demonstrated in the work of Chaki and Datta [51], Aizatulin et al. [52, 53], and Corin et al. [54].

Fuzz testing of TLS implementations. Given their prominence and importance, the research community has put implementations of the SSL/TLS protocols under close scrutiny in recent years. Fuzzing has been a prominent approach in testing SSL/TLS implementations, where test cases are typically synthesized by applying mutation heuristics on known valid inputs (*e.g.* message sequences and certificates). Beurdouche et al. [55] looked at the problem of libraries mishandling unexpected sequences of messages when implementing support for various ciphersuites, authentication modes and protocol extensions. Brubaker et al. [1] used unguided black-box fuzzing to test client-side validation of X.509 certificates in SSL/TLS implementations. Chen et al. [2] extended this approach by using Markov Chain Monte Carlo sampling to guide test case generation, achieving better code coverage with less number of test inputs. De Ruiter et al. [56] showed that the implemented state machine of SSL/TLS can be inferred by applying a fuzzing-based technique, which can then be verified manually to discover errors. A recent work by Somorovsky [57] presented a framework that allows developers to evaluate the behavior of TLS servers in a flexible manner, with the ability to create arbitrary protocol flows and dynamically modified messages.

Differential analysis. In the absence of a test oracle that always generates correct outputs given any possible inputs, one can leverage the principle of *differential analysis* to analyze non-trivial semantic correctness properties of an implementation [58]. For example, X.509 certificate chain validation logic has been investigated before by combining differential testing with fuzzing [1, 2] and with symbolic execution [9]. Differential testing with fuzzing was also used for analyzing semantic correctness of TLS implementations [55].

2.4 Practical attacks

Many weaknesses in the design and implementation of security-critical protocols lead to practical attacks. Here we provide a brief overview of different types of attacks.

i. Attacks against implementations of standardized cryptography

Side channel attacks. Depending on their implementations, cryptographic software might leak secrets through various side channels, which can sometimes be exploited. Several implementations of AES are known to be vulnerable to timing side channel attacks [59, 60]. Similarly, there are exploitable side channels found against implementations of RSA and ECDSA [61–64]. Side channels can still exist even if one uses a trusted execution environment (TEE) like SGX [65, 66].

Padding oracle attacks. Another class of attacks against cryptographic software is aimed at exploiting padding oracles. In essence, this takes advantage of the observable differences in how a victim software handles malformed padding (*e.g.* with special error messages) and other operational failures. For example, Bleichenbacher found that some implementations of the PKCS#1 v1.5 algorithm exhibit an exploitable padding oracle [67], the attack of which was later extended by Bardou et al. [68]. Some implementations of the Cipher Block Chaining (CBC) mode of block cipher operation were found to be vulnerable to padding oracle attacks as well [69].

Signature forgery attacks. Flaws in implementations of PKCS#1 v1.5 RSA signature verification can lead to variants of signature forgery attacks, especially when a small public exponent is being used [7, 8, 11–15]. In many cases, the flaws were due to some unwarranted leniencies in the parsing of RSA output during signature verification, which we will discuss further in Chapter 5.

TLS and other protocols. Since TLS relies on many different cryptographic algorithms, some of the attacks against implementations of such algorithms can also be adapted to exploit TLS. For example, the CBC padding oracle attack is known to work on older versions of TLS (SSL) [69] and in tandem with a downgrade attack [70]. The Lucky Thirteen attack [71] exploits a timing side channel in the MAC-then-encrypt design of earlier versions of TLS to decrypt arbitrary ciphertext, and some pseudo constant time patches aimed at mitigating such side channel are found to be inadequate [72].

In some cases, multiple weaknesses can be exploited together. For example, the KCI attack [73] is made possible due to the use of certain non-ephemeral ciphersuites, plus the fact that installing end-entity (in contrast to CA) certificates do not trigger any warnings on certain systems, and many implementations are not correctly handling the key usage extensions. This also highlights why correctly handling X.509 extensions when validating a chain of certificates, as we will investigate in Chapter 4, is an important matter.

There are also reported attacks against Apple iMessage [74] and 4G LTE [75], both exploiting the lack of authentication guarantees in the counter (CTR) mode of AES, where ciphertexts are known to be malleable.

ii. Against content distribution applications

Cryptanalysis of proprietary algorithms. Given that many early content protection systems use non-standard cryptography, one possible line of research is to perform cryptanalysis. Biryukov et al. [76] present an analysis of the weak cipher (PC1) employed by Kindle for content protection. The authors have shown that due to the lack of avalanche effect in PC1, one can extract the secret key using known plaintext and ciphertext attacks. Crosby et al. [77] present a cryptanalysis on the High Bandwidth Digital Content Protection (HDCP) scheme, an identity-based cryptosystem used for communication in the Digital Visual Interface (DVI) bus. Their analysis shows that given access to 40 public/private key pairs, one can essentially break all the security guarantees promised by the scheme.

Memory dumping. Another possible attack to bypass the cryptographic protection mechanisms employed by content distribution applications is to directly dump the decrypted contents from system memory, as demonstrated by Wang et al. [78]. The authors proposed an approach for identifying data paths of cryptographic operations used by the target applications, and then dump and reconstruct the streams of decrypted contents found in the memory. Such an attack is real-time in nature (*i.e.*, to extract 2 hours worth of content, the attack needs to accommodate a 2-hour long playback). In Chapter 3 we discuss possible attacks against other aspects of content distribution applications, which in some cases lead to a much more efficient content extraction.

3. EVALUATING ROBUSTNESS OF CONTENT DISTRIBUTION APPLICATIONS ON ANDROID

3.1 Introduction

The ubiquity of mobile devices has encouraged *content owners* (e.g., publishing houses and record labels) to tap into the online business ecosystem in an attempt to reach a larger number of audience. As a result, they often retain the service of software developing *content distributors* to adapt to this emerging trend of customer engagement. The role of content distributors is focused on developing mobile applications (*apps*) tailor-made to fit the form of the contents and the business model of content owners, and providing continuous technical support in updating and distributing digitized contents (e.g., magazines and music). For maintaining their business edge it is crucial for the content distributors to ensure that the end users cannot easily have unfettered access to the raw, high-quality reproduction of contents in their devices, even in the cases where the digital contents can be consumed without Internet connectivity (e.g., offline playback). *The overarching goal of this research is to systematically identify (and, in the process, educate developers about) design weaknesses in content delivery apps that can grant users unauthorized and unrestricted access to the underlying content.*

At a first glance, it may seem that the design of an effective *content protection* mechanism boils down to effectively enforcing *Digital Rights Management* (*DRM*). We however argue that there is a subtle distinction between the two. DRM enforcement is concerned about regulating user's access to the content *after* the content (and, the corresponding usage control policy and other bootstrapping information) has been securely delivered to the user's device. On the other hand, content protection mechanisms, especially in the context of online content distribution, *also* have to guarantee reliable receipt of payments and secure delivery of contents (and the accompanying control policies) to the user device.

In this research, we demonstrate that, in many cases, an adversary (the device owner in our context) can modify the enforcement policy[1] while it is being delivered to the device during bootstrapping, to achieve unfettered access to raw contents.

The challenges of effective content protection enforcement is further exacerbated by the fact that content distributors, in order to increase their audience reach, often need to support a plethora of (legacy) devices running various (legacy) versions of operating system (OS). Even after successful bootstrap, effectively enforcing DRM, without considering the *analog loophole problem*—the Achilles' heel of any DRM systems—is a challenging ordeal and requires content distributors to conceal secret states in a potentially hostile execution environment. In different systems, secret states manifest in various aspects of the underlying mechanisms, for example the content encryption keys, authorization tokens, subscription status, or even the raw content itself. The general consensus is that effective DRM enforcement is feasible on a device equipped with a trusted execution environment (TEE). Technologies like the ARM TrustZone have been available on the hardware architecture level for some years now. Nevertheless, various system-on-chip (SoC) vendors have come up with different TEE implementations that do not seem to conform to the same API standard [79]. Together with the fact that TEE vendors often adopt a tight admission control model, currently it is still somewhat difficult to develop widely deployable apps that uses TEE for DRM needs, especially on relatively low-end and legacy devices that do not have the trusted images preloaded. Due to this lack of a generic secure solution that is applicable to all ⟨device, OS⟩ pairs, developers often resort to a best-effort, *deterrence-based enforcement* of DRM—specialized for each ⟨device, OS⟩ pairs—where the main objective is to raise the bar for mounting successful bypass attacks instead of providing

[1]Readers may question the rationale of delivering the usage control policies during app bootstrap, instead of having them hardcoded inside the apps. Delivering policies during bootstrap allows for flexible customization of various aspects of the policies (for instance, number of free trials allowed), to better fit the business decisions.

absolute enforcement guarantees. Since content protection is only as strong as the weakest links (*e.g.*, legacy ⟨device, OS⟩ pairs) in the ecosystem, this presents an interesting trade-off between audience reach and the strength of content protection.

Content Protection Enforcement Analysis: In this research, we systematically identify the different attack surfaces a content delivery app may expose, and how those can be exploited by adversaries to bypass protection enforcement. In our analysis, we consider two abstract classes of adversaries, namely, the *network adversary* (who can observe and manipulate network traffic) and the *local adversary* (who can access internal states and possibly tamper with the execution environment). For each of the two classes, we further consider varying degree of adversary capabilities, ranging from a normal tech savvy user to more sophisticated ones like rooting the device and TLS interception capabilities. With respect to our hierarchy of adversaries, we present concrete attacks against 141 Android content delivery apps, including some high-profile ones like the *Amazon Music*, *Bloomberg Businessweek+*, and *Forbes Magazine* apps. Our evaluation reveals a bleak state of the affair. We observed that all these apps are susceptible to our attacks due to unjustified trust assumption on the underlying technologies, *e.g.*, insecure bootstrapping and policy delivery, and bad practices like client-side policy enforcement, and reuse of content encryption keys. Whenever possible, we further dissect the weaknesses that our attacks exploited and categorize them using the Common Weakness Enumeration (CWE)[2]. We believe that, with the patterns provided by our concrete analysis, this work lays a solid foundation for further research on automated vulnerability detection and the development of more robust apps.

Findings: In our evaluation of publication apps, a somewhat uncharted territory for academic studies, many apps are not only falling short in terms of content protection, but contain weaknesses that allow remote exploits which threaten the app users' security and the privacy. Notable among our findings are the *purchase bypass attacks* against the Forbes Magazine and Mother Earth News apps which allows an

[2]`https://cwe.mitre.org/`

adversary, with sufficient filesystem permission, to manipulate the purchase status for gaining unauthorized access to all issues for free. Another example of relying only on client-side enforcement of usage control policies was exhibited by the Bloomberg Businessweek+ app, for which a network adversary with TLS interception capability can rewrite the policies on the fly during app bootstrap, and obtain *virtually unlimited free previews* of its subscription-only articles. We observed that the service of a content distributor often gets retained by a diverse group of publishers. Since apps from the same distributor tend to be similar in their designs, our attacks affect a large number of different publications.

One popular choice for DRM enforcement is to employ a cryptographic cipher to encrypt the underlying content, in which case the robustness of the enforcement hinges on the concealment of the secret key, as the Kerckhoffs's principle mandates. This approach was embraced by the Amazon Music app, which was the most robust app analyzed in our evaluation. While the app seems to be programmed with the good practice of minimizing exposure time of cryptographic keys in memory, we were still able to devise a *key extraction attack* to extract the underlying content encryption keys by leveraging some non-complex binary instrumentation. To our surprise, a closer inspection of the app revealed that against best practices, the entire Amazon Music collection of 40 million songs seems to be encrypted under one single content encryption key, irrespective of accounts, device models, and subscription tiers. Our successful key extraction attack hence puts their entire collection in serious jeopardy.

Given that our findings could potentially affect the business of various stakeholders, we have engaged in responsible disclosure (Section 3.5.1), and careful ethical considerations have been taken during and after our experiments (Section 3.5.2).

Contributions: In summary, this research makes the following contributions:

1. We identify attack surfaces and practical adversaries with varying degree of sophistication that vendors of content distribution mobile apps should consider, in order to devise effective content protection mechanisms, especially in the

absence of a trusted execution environment (TEE) supported by the underlying platform.

2. We systematically evaluate 141 content distribution apps developed for Android with respect to our identified hierarchy of adversaries. Our evaluation uncovered that more often than not developers of these apps make unjustified trust assumptions about the underlying platform and design decisions that enable an adversary to circumvent these protection mechanisms without having to reverse engineer the apps to extract the underlying secret state.

3. We dissect and classify the weaknesses that our attacks exploited with respect to CWEs to help future developers avoid the same pitfalls. We have responsibly shared our analyses with the corresponding content distributors. With the understanding of various attack strategies, we discuss possible countermeasures, their trade-offs and implications.

3.2 Scope

We now discuss the attack surfaces, threat model, and platform that we consider in this research.

3.2.1 Attack Surfaces

Without loss of generality, in the following discussions, we consider encryption is used to protect the underlying contents. The normal operation of a content distribution app can be roughly broken down into the following six phases: (1) Bootstrapping; (2) Storing authorization token; (3) Content transmission and storage on the device; (4) Playback preparation; (5) Content decryption; (6) Content playback. Each of the above steps presents opportunities for the attacker to bypass content protection and thus corresponds to one of the following attack surfaces. Depending on the actual implementation, certain steps might be skipped or merged, and events might happen in a slightly different order. In this research we focus on **(AS1–4)**.

(AS1) Bootstrapping. In the bootstrapping phase, the app authenticates itself and the user to the content distributor's back-end server, obtains a list of available contents and their prices, as well as authorization to access them. This step may involve monetary transactions, if this is the first time that a user subscribes to the service or purchases contents. Successful completion of the bootstrapping process may result in the back-end server returning an authorization token. Information inside the token can be as simple as URLs of contents, or it could contain rich policy enforcement details including expiration date and maximum playback times to allow granular control. In some cases, it may also contain the content decryption key to allow future consumption of contents. One might attack this surface in an attempt to get the content source URLs or to trick the app with more permissive policies by rewriting the authorization tokens.

(AS2) Authorization token storage. The token may need to be stored on the device's storage to accommodate content access, especially when the business model allows offline playback of contents (without Internet connectivity). An adversary with adequate storage access privilege might be able to retrieve and modify the authorization tokens on the device's storage and gain unauthorized access to contents.

(AS3) Content transmission and storage. Upon receipt of an authorized request, the back-end distribution server sends the content over the Internet. If the content is not adequately protected in transit, an adversary might be able to intercept the communication and duplicate the raw content.

Once the content arrives on the user side, it may get consumed and removed almost immediately, or it might be stored for offline consumption, which most services tend to offer for better user experience, but opens up the possibility for adversaries with sufficient storage access privilege to conduct content extraction attack.

(AS4) Playback preparation. When a user initiates playback of an encrypted content from the device's storage, the app might perform certain control checks (*e.g.*, authorization expiration) and then load the relevant secret keys into the device's memory, so that content can be decrypted for consumption. This presents an opportunity for adversaries who can inspect the device's memory to perform key extraction attacks.

(AS5) Content decryption. During actual playback, (fragments of) the content would need to be decrypted in memory. An adversary who has the capability of inspecting the device's memory can exploit this opportunity to extract the content fragments in clear, and attempt to chain them back into the original content. An attack against this surface given video contents of high entropy has been demonstrated to be feasible [78]. Attacking this surface usually requires real-time effort, that is, to extract 2 hours worth of content, the attack needs to accommodate a 2-hour long playback.

(AS6) Analog loophole. One inevitable attack surface is the analog loophole, where analog signals of protected contents are recaptured during playback. For contents like publications and motion pictures, it can be quite costly to produce high quality replicas. Similar to **(AS5)**, such an attack is also real-time in nature.

3.2.2 Platform and Test Setup

In our studies, we focus on the Android platform because 1) it is the most popular operating system to date and is increasingly the platform where most multimedia contents are being consumed; and 2) there exists a wide range of legacy devices that lack new hardware-enforced isolation and are running old versions of Android. For the different levels of local adversary capabilities, we leverage rooted Android phones running Android version 4.4 (Kitkat) and 5.0 (Lollipop), whichever satisfies the minimum requirements of the studied apps. To emulate network adversaries, we leverage a Linux setup hosting a wireless AP and running MITMProxy[3]. Note that since this research is not about the robustness of Android itself but the content delivery apps that run atop of Android, we deliberately choose older versions of Android devices that are representative of the "weakest link" in the business ecosystem, as would a rational attacker do. This allows us to demonstrate the perils of service providers not excluding such devices from accessing their content distribution services.

Since in all our evaluations, content distributors maintain the back-end distribution servers and develop the Android apps that interact with users and enforce content protection, in the rest of this chapter we use *vendors*, *developers* and *content distributors* interchangeably. When we say *attacks*, we mean that an adversary is able to obtain contents in a manner that violates the control mechanisms in place.

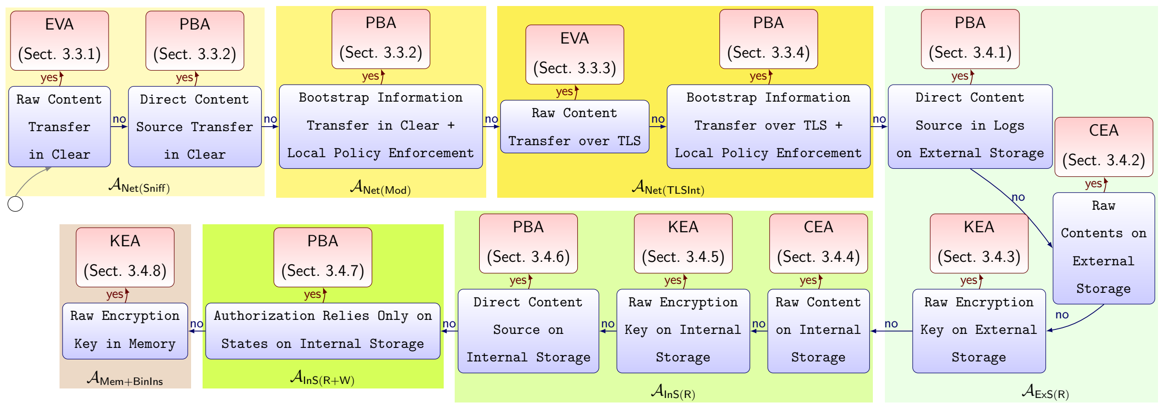
3.2.3 Threat Model

Meaningful discussion on robustness of access control and protection mechanisms requires a well-defined adversary model which bounds attackers' capabilities. Here we discuss the two categories of capabilities that we consider in this research, focusing on software-only attacks. An enumeration of successful attacks and the corresponding adversary capabilities can be found in Figure 3.1.

Network Adversaries

$\mathcal{A}_{\mathsf{Net(Sniff)}}$ **(Passive Eavesdropping of Network Traffic)**. Such an adversary enjoys the capability of passively observing the network traffic between the device and the back-end servers serving the app. We also assume the adversary is capable of extracting payloads out of the network packets being observed and parsing messages of standard plaintext protocols (e.g., HTTP). This represents the lowest capability among all the $\mathcal{A}_{\mathsf{Net}}$ adversaries.

[3] `https://mitmproxy.org/`


EVA (Sect. 3.3.1)
yes
Raw Content Transfer in Clear
no
PBA (Sect. 3.3.2)
yes
Direct Content Source Transfer in Clear
$\mathcal{A}_{\text{Net(Sniff)}}$
no
PBA (Sect. 3.3.2)
yes
Bootstrap Information Transfer in Clear + Local Policy Enforcement
$\mathcal{A}_{\text{Net(Mod)}}$
no
EVA (Sect. 3.3.3)
yes
Raw Content Transfer over TLS
no
PBA (Sect. 3.3.4)
yes
Bootstrap Information Transfer over TLS + Local Policy Enforcement
$\mathcal{A}_{\text{Net(TLSInt)}}$
no
PBA (Sect. 3.4.1)
yes
Direct Content Source in Logs on External Storage
no
CEA (Sect. 3.4.2)
yes
Raw Contents on External Storage
no
KEA (Sect. 3.4.3)
yes
Raw Encryption Key on External Storage
$\mathcal{A}_{\text{ExS(R)}}$
no
CEA (Sect. 3.4.4)
yes
Raw Content on Internal Storage
no
KEA (Sect. 3.4.5)
yes
Raw Encryption Key on Internal Storage
no
PBA (Sect. 3.4.6)
yes
Direct Content Source on Internal Storage
$\mathcal{A}_{\text{InS(R)}}$
no
PBA (Sect. 3.4.7)
yes
Authorization Relies Only on States on Internal Storage
$\mathcal{A}_{\text{InS(R+W)}}$
no
KEA (Sect. 3.4.8)
yes
Raw Encryption Key in Memory
$\mathcal{A}_{\text{Mem+BinIns}}$

§ KEA = Key Extraction Attacks; PBA = Purchase Bypass Attacks;

CEA = Content Extraction Attacks; EVA = Eavesdropping Attacks.

In this research, KEA and EVA both imply CEA.

Fig. 3.1.: Enumeration of possible weaknesses and attacks under various adversary capabilities in attack tree form

$\mathcal{A}_{\mathsf{Net(Mod)}}$ **(Active Modification of Network Traffic)**. The adversary can modify and selectively block both incoming and outgoing network traffic, in order to change what the target apps receive. For plaintext protocols without strong integrity and authenticity guarantees, such adversary is also able to modify the content of protocol messages undetected. This is easily attainable by deploying a proxy server.

$\mathcal{A}_{\mathsf{Net(TLSInt)}}$ **(Interception of TLS Traffic)**. This is an upgrade to $\mathcal{A}_{\mathsf{Net(Mod)}}$ with the added capability of intercepting encrypted TLS traffic, as done quite frequently by anti-virus and parental control software [18], as well as middle-boxes in enterprise settings [19]. On top of a proxy setup, exactly how to attain this capability depends on the actual implementation. For target apps that trust the system CA store, it could be as simple as importing a new CA certificate into the trusted CA store as an unprivileged user. For apps that trust only their own CA stores or use key pinning, one might need the help of $\mathcal{A}_{\mathsf{InS(R+W)}}$ or even $\mathcal{A}_{\mathsf{Mem+BinIns}}$, both of which are discussed below.

Local Adversaries

$\mathcal{A}_{\mathsf{ExS(R)}}$ **(External Storage Read Any)**. This adversary capability can be achieved by a device user who has the minimum technical sophistication necessary for accessing and transferring files available on external storage of an Android device, which is "*world-readable*" [80] without any special modifications to the device. Storing large downloaded files on the external storage is a common practice in order to cope with devices that have internal storage of very limited size. On a side note, the two storage areas of Android are named *internal* and *external* due to historical reasons, and even on a device without actual physically removable media, the external storage area would still exist [80].

$\mathcal{A}_{\mathsf{InS(R)}}$ **(Internal Storage Read Any)**. Such an adversary has the privilege to read arbitrary files on the internal storage of the device. A mobile OS such as Android[4] usually provides isolation so that an app can only read its own internal storage, and a normal user is by default not given direct access to the system's internal storage. Consequently, this capability is usually attained by "rooting" the device.

$\mathcal{A}_{\mathsf{InS(R+W)}}$ **(Internal Storage Read Write Any)**. We consider this adversary to have the capability of reading and writing any files to any location of the internal storage. Though rooting the device would typically grant permissions to both read and write access to the internal storage, we make this fine-grained differentiation for the sake of generality, as each of these capabilities can enable different attacks.

$\mathcal{A}_{\mathsf{Mem+BinIns}}$ **(Memory Inspection and Binary Instrumentation)**. The final adversary we consider is the most powerful one in the software domain without tampering hardware. This capability not only allows the inspection of the target app's internal execution state in memory but also the modification of the execution (control flow) of the app through binary instrumentation.

3.2.4 App Selection

Our evaluations start with manual analysis of some representative apps. Then, with the initial findings, we try to automate our attacks, and collect more apps that follow similar designs, and automatically test whether they are also vulnerable.

We chose the *Amazon Music* app because it is well-known and popular in the streaming business. At the time of writing, it has more than one hundred million installs and was one of the top 10 *"Music & Audio"* apps on the Google Play store. After successfully devising an attack, we then recreated it against the *Audible* app,

[4]Given that the user who did the installation get to choose the administrator/sudo password, conventional desktop operating systems like Windows and Linux don't have such a separation of storage space. In general, administrator/sudo privilege allows one to perform memory inspection and binary instrumentation.

which is another highly popular app also owned by Amazon, and the 2 apps happened to be using a very similar implementation.

We then focus on the publishing industry. We picked the *Forbes Magazine* and *Bloomberg Businessweek+* apps, as they are both well-known and popular business publications, which were coincidentally made by the same developer using 2 different designs. Having studied apps of US-based magazines, we then switched to look at their counterparts from the UK. We chose *Cosmopolitan* and *ELLE* as they are well-known magazines. We then collected many other publication apps that follow similar designs, to show that the weaknesses we found are indeed affecting a wide range of publishers and their publications. Finally we found a few publication apps that exhibit different weakness patterns on the lower-end of the spectrum, completing the study.

We give the full list of apps studied in this research in Table 3.1 at the end of this chapter. A vendor might use several different designs for its content distribution apps. In the rest of this chapter, apps that are using similar designs (and hence susceptible to the same attacks) are grouped and discussed together.

3.3 App Weaknesses & Network Attacks

Here we present the weaknesses we found in the studied apps, as well as concrete network attacks that exploit them. We note that some weaknesses in this section also pose threats to the app users' security and privacy.

As an effort to systematize our findings, for most known weakness patterns, we map them to the relevant Common Weakness Enumerations (CWEs) in our analysis. A list of all the CWEs discussed in this research can be found in Table 3.2 at the end of this chapter.

3.3.1 Raw Content Transfer In Clear

If a content delivery app receives its contents in clear, an attacker with $\mathcal{A}_{\mathsf{Net(Sniff)}}$ capability who can passively observe traffic exchanged between the device and the content distribution back-end server would be able to eavesdrop, extract and duplicate contents for free. We found that the *The MagPi*, *Business Money*, *Artists & Illustrators*, *My MS-UK*, *Popshot Magazine* apps (group-1 of Table 3.1) fall into this category. These are apps of magazines from different publishers, all made by a vendor called *Apazine*.

Eavesdropping Attacks. We note that in Apazine's design, contents are distributed based on `PDF`s, with each issue of the magazines and journals encapsulated in a single `PDF` file. Issues of publications can be purchased individually inside the apps, which would trigger a `PDF` download. However, because the apps and the back-end servers exchange data including the unencrypted content PDFs through HTTP (instead of HTTPS) [CWE-319], it is trivial for $\mathcal{A}_{\mathsf{Net(Sniff)}}$, the weakest remote adversary we consider, to extract and duplicate the PDF files through the observed traffic. This is an attack against **(AS3)**. We have confirmed the feasibility of this attack in the aforementioned apps.

3.3.2 Bootstrap Information Transfer in Clear

It is often necessary for publication apps to communicate with the back-end servers to get bootstrapped with information regarding what issues and subscription tiers are available at what price. We note that many apps we studied receive their bootstrap information in clear through HTTP, which is another instance of [CWE-319]. This leads to 2 different attacks on **(AS1)**, given varying levels of adversary capabilities.

Purchase Bypass Attacks with $\mathcal{A}_{\mathsf{Net(Sniff)}}$. The 5 group-1 apps discussed in Section 3.3.1 can again serve as examples, as they are all susceptible to this attack.

From Apazine's back-end server they receive bootstrap information in `JSON` format, which contains details of each issue. Specifically the URLs for downloading the unencrypted content PDF files of each issue can be found there as Base64 encoded ASCII strings. Since given those URLs, the back-end content distribution server does not enforce further authentication and authorization before serving the PDF files [CWE-425], an $\mathcal{A}_{\mathsf{Net(Sniff)}}$ adversary can observe and parse the `JSON`, decode the URLs, and get unrestricted direct access to the unencrypted content PDFs. We verified the feasibility of this attack by observing the traffic generated by the 5 aforementioned apps.

Purchase Bypass Attacks with $\mathcal{A}_{\mathsf{Net(Mod)}}$. There exist other possibilities for exploits even if the bootstrap information does not contain direct content sources. For concrete examples, we turn to the 70 publication apps (group-3 of Table 3.1, *e.g.*, *Forbes Magazine*) made by a developer called *Maz Systems*, which is reported to have an annual revenue of several million US dollars [81]. The design for these apps seems to rely on the apps to construct the content source URLs, based on the bootstrap information received in `XML` format and the unique IDs of each issue. The back-end server hosted on Amazon S3 requires some level of API key authentication before serving the contents. However, since the price of each issue is directly given by the bootstrapping `XML` received through plain HTTP without much integrity and authenticity guarantees [CWE-354], we found that an $\mathcal{A}_{\mathsf{Net(Mod)}}$ adversary can rewrite the price of all the issues into zero, and the 70 apps we tested all trusted their corresponding altered `XML`, and offered magazine issues for free. The adversary can then use the apps to download the publications without paying. Additionally, some publishers offer subscriptions to their publications in the apps (*e.g.*, $29.99 per year for Forbes Magazine), the price of which was also received from the same bootstrapping `XML`. We have confirmed that an $\mathcal{A}_{\mathsf{Net(Mod)}}$ adversary can also rewrite the prices of subscription plans into zero, then subscribe (for free) and get access to all the issues available within the subscription period.

These findings suggest that the price of purchase is enforced by the apps locally on client-side [CWE-603] without involving the back-end servers after the initial bootstrap.

3.3.3 Raw Content Transfer over TLS

Also for those 70 group-3 apps, after a purchase has been confirmed, it receives the contents, in the form of a `ZIP` file, from some back-end server hosted on Amazon S3 through TLS. Despite using encrypted connections, it does not mean one cannot attack **(AS3)**.

Eavesdropping Attacks. Specifically, we found that for establishing a TLS session, those apps trust the system CA store for signing certificates and do not seem to be using any forms of key/certificate pinning. As the result of which, it was trivial to attain $\mathcal{A}_{\mathsf{Net(TLSInt)}}$, without the need to leverage other advanced local capabilities. Together with the fact that the `ZIP` files were not passphrase-protected, an $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ adversary can extract contents out of the passively observed `ZIP` files with ease.

3.3.4 Bootstrap Information Transfer over TLS

Even if apps receive bootstrap information over encrypted TLS connections, without additional integrity and authenticity guarantees, an $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ adversary can still abuse such information for his/her own gains. As concrete examples, we look at *a*) the 34 publication apps (group-4–6 of Table 3.1) exemplified by the *Bloomberg Businessweek+*, *Entrepreneur Magazine* and *Men's Health Magazine* apps, which were coincidentally also developed by *Maz Systems*, under designs different from the group-3 apps; and *b*) the 30 publication apps (group-7–8 of Table 3.1) exemplified by *ELLE UK* and *The Independent*, developed by a vendor called *Pugpig*.

Purchase Bypass Attacks. Similar to the apps discussed in Section 3.3.3, these 34 group-4–6 apps all trust the system CA store, so attaining the $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ capability was straightforward. For the group-5–6 apps that offer periodicals, they receive detailed information regarding what issues are available at what price through some bootstrapping `JSON` over TLS. The description of each issue comes with a boolean indicating whether it is locked (require payment) or not. We have found that using the $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ capability, one can rewrite all instances of `"locked": true` into `"locked": false` in the bootstrapping `JSON`, and have all the issues unlocked for free.

On the other hand, the group-4 apps employ a different, article-centric subscription-based business model, which allows its users to read k number of articles for free every j days as trial. Likewise, the value of both k and j are retrieved from some bootstrapping `JSON` transferred over TLS. We have confirmed that an $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ adversary can rewrite the value of k and j in the bootstrapping `JSON`, as shown in Figure 3.2, to trick the apps into granting everyday a number of free articles so large that it is virtually like having a paid subscription.

Original JSON Snippet

```
... ...,
"metering": {
    "freeViews": 4,
    "resetAfter": 28,
    "registerAfter": 2,
    "registerRequired": false
}, ... ...
```

Snippet After Rewrite

```
... ...,
"metering": {
    "freeViews": 400,
    "resetAfter": 1,
    "registerAfter": 300,
    "registerRequired": false
}, ... ...
```

Fig. 3.2.: Rewrite bootstrapping JSON with $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ to gain free articles in the Bloomberg Businessweek+ app

Additionally, the 30 group-7–8 apps receive from their back-end server a series of `XML` files describing available issues and their pages. With $\mathcal{A}_{\mathsf{Net(TLSInt)}}$, we found that one can parse the `XML` files, stitch various metadata components into the actual content source URLs and download magazine pages directly without paying.

All these findings suggest that the access control enforcement (*e.g.*, locked contents and free trial previews) are done locally on the client-side without involving the back-end servers [CWE-603].

3.3.5 Threats to User Security and Privacy

For the 75 group-1 and group-3 apps discussed in Sections 3.3.1 and 3.3.2, since their bootstrap information are sent in clear without strong integrity guarantees [CWE-354], any Man-In-The-Middle (MITM) can easily tamper with what is being transferred. This not only allows one to bypass purchase and extract contents, but also poses threats to the app users. For example, one might be able to increase the price of each issue to induce financial losses on the user. One can also remove specific issues in the bootstrap information to implement censorship. Rewriting URLs can also trick the users to visit some potentially malicious websites. Additionally, given known vulnerabilities about the libraries that the apps uses (*e.g.*, MuPDF [82], Zip [83]), one can potentially change the URLs in the bootstrap information to point to some maliciously crafted input files to attack the user's device.

Furthermore, for many of the 70 group-3 apps discussed in Sections 3.3.2 and 3.3.3, we have observed that during and after the bootstrap, some tracking data are being sent to the back-end over HTTP in clear. The exchanged data contains the device unique identifier and model name, along with some session ID and publication ID. A passive eavesdropper might try to extrapolate who is reading what magazines, which could be quite revealing given that some publications are related to medical conditions, musical instruments and specific industries, posing threats to the app users' privacy.

3.4 App Weaknesses & Local Attacks

Here we present more weaknesses of the studied apps, with a focus on local attacks. Similar to the previous section, we map our findings to the relevant CWEs whenever possible.

3.4.1 Log File Leakage

Another possible weakness is leakage of secrets through log files, similar to what had previously been observed in some Android mobile banking apps [29].

Purchase Bypass Attacks. As discussed previously in Section 3.3.1, those 5 group-1 apps use direct content source URLs for fetching contents. Our inspection revealed that those same apps leave some debugging log files on the external storage which contain both the direct URLs of publication PDF files hosted on their back-end servers and the identifiers of each of the issues available for purchase [CWE-532]. This allows an $\mathcal{A}_{\mathsf{ExS(R)}}$ adversary to retrieve those URLs, and by replacing the appropriate portion of the URLs with the issue identifiers, one can enumerate the different published issues and download their corresponding unencrypted PDF files directly [CWE-425], effectively getting unlimited unauthorized access without having to purchase, mounting an attack against **(AS2)**.

3.4.2 Raw Content on External Storage

If the apps leave their contents on the External Storage, it would allow for an easy attack on **(AS3)** that both the apps and the publishers would lose control of the contents.

Content Extraction Attacks. We have found that the 9 group-6 apps serve contents in the form of `PDF` and put their `PDF` files on the device's external storage. Given the $\mathcal{A}_{\mathsf{ExS(R)}}$ capability, one can easily get those files and make copies of them.

This can be applied to the various free trial issues offered in the apps, as a user is allowed only several minutes of free preview before needing to pay to continue reading, but one can simply workaround this restriction by copying the full `PDF`s from the external storage and open them using a different reader.

3.4.3 Raw Encryption Key on External Storage

Even if an app employs encryption as the means for content protection, if the secret key is left in a place that is accessible by an adversary, one can attack **(AS2)** and strip the encryption.

Key Extraction Attacks. As examples, we again look at the 5 group-1 magazine apps discussed in Section 3.3.1. After purchasing a specific issue, those apps would download the content file and put it in the external storage of the device. With the $\mathcal{A}_{\mathsf{ExS(R)}}$ capability, we can see that the content files retain the `.pdf` extension but the contents are actually scrambled. Since not even the PDF metadata are comprehensible, we deduce that this is most likely due to the use of a whole file encryption. Together with the findings from Sections 3.3.1 and 3.3.2, this suggests that the content encryption was done locally on the client device after download.

While navigating through the files created by the apps on external storage with the $\mathcal{A}_{\mathsf{ExS(R)}}$ capability, we found that there exists a serialized Java object outside the directory that contains the encrypted PDF files, adjacent to the log files discussed in Section 3.4.1. A quick inspection revealed that this serialized object is of the class `javax.crypto.spec.SecretKeySpec`, which turns out to contain the secret key used to encrypt the PDF files. That object also revealed that the encryption algorithm used was AES, though the exact block cipher mode remains unclear. This is tantamount to leaving one's house key under the doormat outside the house, a known weakness pattern described by [CWE-313] and [CWE-921].

After identifying the key, decrypting the content PDF files was somewhat straightforward. With around 200 lines of Java code and some trial-and-error to determine that the apps were using the Electronic Codebook (ECB) mode of AES, we confirm that the contents can be decrypted using the suspected secret key. We have verified this attack with a paid purchase of a recent issue in the My MS-UK app, and free trial issues in the Business Money, Artists & Illustrators, The MagPi and Popshot Magazine apps.

3.4.4 Raw Content on Internal Storage

Given that the official Android development training material claims files stored on internal storage are "*accessible by only your app*" and "*neither the user nor other apps can access your file*" [80], it is perhaps unsurprising that some apps are making strong assumptions about the confidentiality guarantees provided by the internal storage. Such assumptions, however, can be invalidated with $\mathcal{A}_{\mathsf{InS(R)}}$.

Content Extraction Attacks. For concrete examples, we again look at the 70 group-3 apps discussed in Sections 3.3.2 and 3.3.3. In their designs, each page of the publication is a `JPEG` image of about 0.7 megapixel. After downloading the content `ZIP` file of an authorized issue, the app extracts from it the content images and have them stored on the app's internal storage. The app then acts like an image viewer for displaying each page for the user to read. As the images of each issue are left inside the internal storage without further scrambling [CWE-313], an $\mathcal{A}_{\mathsf{InS(R)}}$ adversary can easily access and make copies of the magazine issues, attacking **(AS3)**.

Through the in-app free previews, we found that the 16 group-8 apps also has each page of an issue saved as a `JPEG` image on the internal storage. In fact, we found that even though the free previews should allow only a small number of pages, all the other pages of the selected issue are already downloaded. Consequently, with $\mathcal{A}_{\mathsf{InS(R)}}$, one can easily bypass the preview limit and access the saved pages directly.

3.4.5 Raw Encryption Key on Internal Storage

Similarly, developers might put encryption keys on the internal storage assuming confidentiality [CWE-313], however, in the face of the $\mathcal{A}_{\mathsf{InS(R)}}$ capability, such a design manifests into an exploitable weakness on **(AS2)**.

Key Extraction Attacks. We use the Counter Intelligence Plus app (group-2) as an example, which is also made by *Apazine*. Interestingly, despite being older than the other group-1 apps discussed before, the Counter Intelligence Plus app appears to be doing a slightly better job in terms of hiding the secret key used in content encryption. In this case, instead of putting it on the "world-readable" external storage, the key is stored on the device's internal storage. However, with the $\mathcal{A}_{\mathsf{InS(R)}}$ capability, we have managed a key extraction attack similar to what is described in Section 3.4.3.

3.4.6 Direct Content Source on Internal Storage

Leaving direct links to contents that do not enforce authentication and authorization [CWE-452] on Internal Storage is another exploitable weakness on **(AS2)**.

Purchase Bypass Attacks. With the exception of the *The Rebel Media* app, all the other 24 group-4–5 apps that offer articles (*e.g.*, the *Bloomberg Businessweek+* app) or video clips (*e.g.*, the *Outside TV Features* app), leave direct URLs to their corresponding contents on the apps' Internal Storage, organized by the different issues, allowing an $\mathcal{A}_{\mathsf{InS(R)}}$ adversary to easily crawl for those and access contents without paying.

3.4.7 Client-Side Authorization

The assumptions on internal storage indeed presents an interesting attack vector. In addition to confidentiality, one might also assume that the internal storage provides strong integrity guarantees. Such an assumption can be invalidated with $\mathcal{A}_{\mathsf{InS(R+W)}}$.

Purchase Bypass Attacks. Each of the 70 group-3 apps discussed in Section 3.3.2 also keeps a local database of published and available issues on the app's internal storage. For each issue, the database keeps a record about the name and date of the issue, a brief description, price, and some other metadata, including the purchase status. With the $\mathcal{A}_{\mathsf{InS(R+W)}}$ capability, we have verified that one can modify the database and replace the default value of the purchase status column with some appropriate values [CWE-642] to trick the app into granting access to magazine issues that were not paid for.

This shows that the authorization of those apps is localized and done unilaterally on the client's device, and does not involve the back-end content distribution server [CWE-603]. Consequently, the robustness of such authorization mechanism hinges on the assumption that the internal storage guarantees integrity [CWE-654], which does not hold given an $\mathcal{A}_{\mathsf{InS(R+W)}}$ adversary.

3.4.8 Raw Encryption Key in Memory

Even if raw secret keys are not left in the clear on permanent storages, they might be loaded into the memory, which presents another opportunity for attacking **(AS4)**. As a concrete example, we look at the Amazon Music app (version 6.5.3), which offers both streaming and offline playback of music to its subscribers. There are two tiers of subscription: Amazon Prime and Amazon Music Unlimited, with the only difference being the size of the collection accessible (2 million versus 40 million songs). Both tiers allow subscribers to download music available from their corresponding collections for offline playback. The downloaded songs are stored on the external storage of the Android device.

Storage Inspection. A quick inspection of the downloaded files shows that regardless of the subscription tier, songs appear to be encrypted and contain a human readable `PlayReadyHeader` XML object in their metadata [84], suggesting that this entire streaming service is using the Microsoft PlayReady DRM framework. With the contents already available on the external storage, we choose to focus on devising a key extraction attack.

We first leverage the $\mathcal{A}_{\mathsf{InS(R)}}$ capability to inspect the internal storage and see if one can attack **(AS2)**. As there exists quite a few secret key candidates (*e.g.*, Base64 strings that decode into binary values of various lengths), we soon run into the problem of not knowing how to verify whether a key candidate is the right one for content decryption.

Key Verification Oracle. Fortunately, the limited documentation publicly available regarding the `PlayReadyHeader` [84] turns out to be quite useful. The `PlayReadyHeader` metadata object contains the key ID, content encryption algorithm (in this case AES CTR mode) and the key length (16-byte), so we know what we are searching for. Better yet, it also contains a checksum used by the framework to protect against mismatched keys. According to the documentation, this is meant to prevent the case where decryption is done with an incorrect key, the subsequent output of which might damage audio equipments during playback. Since the checksum is simply the first 8 bytes of encrypting the 16-byte key ID with the key in AES ECB mode, which is easily computable, we now have an oracle for verifying key extraction correctness, without having to rely on decrypting the contents themselves.

A quick trial-and-error showed that none of our initial suspects were the right content encryption key. The publicly available documentation regarding the PlayReady framework [85] suggests that the content decryption keys are contained inside licenses. We then turned our attention into finding the license instead. We realized that there exists a `.hds` file, which according to some discussion about Silverlight [86], seems to contain the licenses. Without documentations on how to parse and interpret this file, the format of licenses, and whether this file is obfuscated or encrypted, key extraction from it seems could be quite complicated.

Key Extraction Attacks. Instead, we switch our focus to attack surface **(AS4)**. The intuition is that, even if the local license store on internal storage has complex protection mechanisms in place, the content encryption key would still need to be read from the license store and might be loaded into the memory in clear. Hence we upgrade the adversary capability to $\mathcal{A}_{\mathsf{Mem+BinIns}}$, by using the Frida[5] dynamic instrumentation framework. While we were able to trace the app's file read operations by hooking the `read()` system calls with $\mathcal{A}_{\mathsf{Mem+BinIns}}$, including those that reads from the license store, it remains unclear how to interpret the bytes being read. Since tracing and interpreting the preparation phase seems to be quite messy, we take a slightly different approach. With the intuition that sensitive information (*e.g.*, content encryption key), if they were indeed loaded into the memory, might exhibit some recognizable structure and would need to be released at some point (*e.g.*, after content playback), we try to hook deallocation functions instead. While it might also work to hook the `free()` function calls, a quick inspection of the native libraries used by the app shows that they are exporting some functions that seem to be used for deallocating sensitive information. Intercepting the entrance to some of those functions, tracing appropriate pointers, and then dumping a large enough portion of memory pointed by those, we successfully extracted the content encryption key, which verifies against the oracle discussed earlier.

Unlike in previous work where PlayReady protected video contents were reported to be partially encrypted [78], in this case we have observed that the entire original content is encapsulated in a so-called envelope file. Though we were unable to obtain documentations regarding the metadata (besides the `PlayReadyHeader` object), based

[5]`https://www.frida.re`

on some header files publicly available on Github [87], we were able to guess and parse the metadata correctly. In the end it took us about 260 lines of Java code to parse the envelope file and decrypt with AES CTR using the extracted secret key to get the raw audio tracks out.

One might also wonder why $\mathcal{A}_{\mathsf{Mem}}$ alone is not strong enough. We note that the key extraction problem has a two-dimensional search space, spanning memory layout and time. While ultimately it is the memory inspection that gets us the key, without binary instrumentation, however, it is difficult to pinpoint the exact timing, especially if the implementation tries to minimize key exposure by actively releasing and overwriting memory regions containing the secret key, a practice also recommended by previous work [88]. Not knowing when to dump the memory would make it hard for $\mathcal{A}_{\mathsf{Mem}}$ to extract the key. Interestingly, in this app, we have observed a behavior that sensitive information deallocation happens as early as the actual music playback starts. Our speculation is that, since the CTR mode generates keystream blocks by encrypting the next counter values, after a long enough keystream has been generated to allow decryption of the entire content, the app removes the key from memory as soon as possible to minimize exposure time. This is exactly why $\mathcal{A}_{\mathsf{Mem+BinIns}}$ has an advantage in reducing uncertainties along the time dimension.

We further found that the Audible app (version 2.25.0) is also susceptible to this attack. Specifically, members are offered premium podcasts that can be downloaded for offline playback, in which case they are encrypted in the same manner as songs in Amazon Music. It appears that the PlayReady implementations in the 2 apps are quite similar, as our key extraction attack also worked. Since the downloaded Audible podcast tracks are partially encrypted `isma` files, instead of using our decryption code for Amazon Music, we used the Bento4 tookit[6] for successful decryption.

Key Scanning Heuristics. Curious readers might wonder how we recognized the 16-byte key from memory dumps. As explained in previous work, besides loading

[6]`https://www.bento4.com/`

the raw secret key into memory, many cryptographic implementations speed up computation by precomputing the key schedules made of the different round keys [88]. This is because typical block ciphers, including AES, go through multiple rounds of operations to encrypt/decrypt a block, and each round involves a round key derived from the raw secret key. Having to repeatedly expand the raw key into the same key schedules for each block could be quite inefficient. It turns out that the key schedule observation also applies to this particular PlayReady implementation. Using the `keyfind` program [88], we were able to confirm the mathematical relation between the suspected raw key and the derived round keys, strongly suggesting that those bytes found in the memory dump indeed constitute a key schedule.

One key to rule them all. Based on the handful of songs that we sampled in our proof-of-concept experiments, Amazon Music appears to be reusing content encryption keys across songs and different accounts, despite the fact that the PlayReady framework allows a much more granular key binding (*e.g.*, per individual item), as noted in a previous work [78]. We speculate that this is to lower the load and management overhead on the back-end servers, though we are not sure whether the entire ecosystem uses only one single key, or are keys different across data centers in various locations. Consequently, songs made available for offline playback from many different albums across artists, regardless of which tier of subscription and user accounts they came from, might all be decrypted with the same key. This puts the whole collection of 40 million songs available on Amazon Music in excessive risk.

While the attack we presented is agnostic to key granularity and can be performed over and over again to exhaust all the possible keys, however, a more fine-grained key binding (*e.g.* per album or even per song) would have at least required more effort from an attacker, and hinder automatic mass decryption of a large number of songs. Sharing keys across many accounts does not seem to be a good practice, as it is easy to have a single key leaked (*e.g.*, by an insider) and cause large damage to the ecosystem. Interestingly, unlike Amazon Music, Audible seems to be much more granular with its content encryption keys. It appears that for each podcast track, a new key is used for encryption.

3.5 Discussions

3.5.1 Responsible Disclosure and Aftermath

We have notified the content distributors of our findings and provided them with sufficient details to understand and reproduce the attacks. In all cases, we have given the vendors more than 90 days before this research is made public.

In response to our initial report sent in Feb 2018 regarding bypass attacks with $\mathcal{A}_{\mathsf{InS(R+W)}}$ (Section 3.4.7), developers at Maz Systems implemented the use of encrypted database on newer versions of some of the group-3 apps. This is a solution that we do not endorse, as it does not change the client-only nature of the authorization mechanism, so the pattern of [CWE-603] still holds. We followed up with reports on group-4–6 apps in Jun 2018. They have expressed gratitude to our efforts, and are working on app improvements.

We sent several reports to Apazine in Feb, Apr and Sep 2018 regarding weaknesses in their group-1–2 apps. They have replied in Sep 2018 suggesting that the magazines are in public domain and do not contain any sensitive or valuable information, so our implied expectation of robustness is not relevant. We point out that if the contents indeed have no market values then not using encryption can improve user experience, and that some publications in print (*e.g.*, Business Money and My MS-UK) do not seem to be available for free.

Developers at Pugpig have been notified in Jul 2018. They have since acknowledged and confirmed our findings, and replied that they are aware of the weaknesses, and that the apps are designed that way by choice to accommodate anonymous sharing of magazine pages, a feature requested by their clients (publishers).

Amazon has been notified about the key extraction attack against the Amazon Music app in Jan 2018. They responded to our report with several new versions of their music app, implementing new obfuscation strategies and offline playback restrictions on rooted devices. However, despite our recommendation of considering the secret key compromised and switching over to a new key, as of Jun 2018, we have noticed that recent new releases are still encrypted using the same old key. The KEA against Audible was reported to Amazon in Jun 2018, and new versions implementing various obfuscation strategies have since been released.

3.5.2 Legal and Ethical Matters

First and foremost, this research is definitely not aimed at assisting piracy. We have not and will not distribute any code and other artifacts used in conducting the experiments.

As this research was done inside the United States, it is our understanding that the DMCA security research exemption [89] should be applicable. We believe what we did in this research meets the four main requirements for the said exemption: 1) the apps and the device of which the apps were running on were all lawfully acquired; 2) the experiments were done *solely for the purpose of good-faith security research*; 3) the research was conducted *in a controlled setting designed to avoid harm to individuals or the public*; 4) the research did not begin before October 28, 2016.

For ethical reasons, after an attack has been demonstrated to be working, we stop our experiments and did not perform mass content extraction for our personal gains. For example, in the case of Amazon Music, we tried content decryption on only four songs, in order to gain confidence that decryption with the same key would work on songs that are: 1) from different tiers of subscription; 2) stored on the same device but different albums; 3) stored on different devices. Similarly, for each of the group-7–8 apps, we only downloaded 2 random pages from 2 magazine issues. The extracted raw contents (*e.g.*, audio tracks and magazine pages) have been subsequently deleted. We have also engaged in responsible disclosure with the app vendors, demonstrating good-faith.

3.5.3 Possible Countermeasures and Challenges

Bilateral Policy Enforcement. While a stateless sever allows for a more simplistic deployment, as [CWE-603] has noted, a client-only authorization is weak and can potentially be bypassed, especially on an environment where execution/code can be reverse-engineered and tampered with. Since local adversaries are not able to directly tamper with the execution state of a remote server, considering the threats of $\mathcal{A}_{\mathsf{InS(R+W)}}$ and $\mathcal{A}_{\mathsf{Mem+BinIns}}$, policy enforcement can be done more robustly involving the back-end servers. For example, to avoid the attacks discussed in Sections 3.3.2, 3.3.4 and 3.4.7. the authorization logic should be shifted to the back-end server. Then client-side modification of prices and purchase status would result in detectable discrepancies with records on the server, and the latter can refuse to serve contents in such cases.

Direct Content Sources. To hinder the attacks discussed in Sections 3.3.2, 3.4.1 and 3.4.6, content source URLs should not be left in a log file [CWE-532] and also not on a storage that an adversary has unlimited access to [CWE-921]. Instead of explicitly saving the URLs, it would perhaps be better to have them constructed dynamically during runtime, and the servers should request extra authentication and authorization. An $\mathcal{A}_{\mathsf{Mem+BinIns}}$ adversary might still be able to figure out the URLs and the accompanying parameters, but it would be an improvement comparing to the current deployments.

Certificate Pinning. To hinder the $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ attacks discussed in Sections 3.3.3 and 3.3.4, instead of trusting the system CA store, the apps could adopt some forms of key/certificate pinning [7]. Even though $\mathcal{A}_{\mathsf{InS(R+W)}}$ and $\mathcal{A}_{\mathsf{Mem+BinIns}}$ might still be used in tandem to defeat pinning, it would at least make $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ more difficult to achieve than in the current implementations.

[7] `https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning`

Denying services to rooted devices. One might propose for the apps to stop providing services on rooted devices, as on unrooted ones the adversary capabilities are greatly limited. This approach however has its own challenges. First, while a concrete global number of rooted devices is not available, it has been suggested that the number could be quite high in certain communities [90,91]. Various rooting tools have reported millions of downloads [92, 93], and the superuser access management app has hundreds of millions of installs [94]. A content distributor denying service on rooted devices risks losing these customers. Second, determining whether a device is "rooted" is an on-going arms race. Depending on the heuristics used and how the checks were implemented, binary instrumentation might be able to bypass those as well [95]. We have observed that, as of version 7.5.4, the new Amazon Music restriction of no offline playback on rooted devices can be bypassed with RootCloak [96], a popular system modification module.

Google has since introduced the SafetyNet service for developers to detect if a device has been tampered with. Android Pay, Netflix, and Pokemon Go are some examples that would deny service if SafetyNet finds the devices is rooted. However, the cat-and-mouse game between SafetyNet and the Magisk systemless rooting technique in 2017 has been well documented [97, 98], and there are reports suggesting that on *legacy Android versions* various bypass and attacks against SafetyNet are possible [99].

Anti-Debugging and Anti-Instrumentation. Another possibility is to implement anti-debugging and anti-instrumentation techniques in the apps to hinder analysis, potentially on even rooted systems. Depending on what heuristics are being used, some might still be bypassable [100]. With the advancements of artifact detection [101], anti-instrumentation [102,103], and transparent debugging [104,105], this line of defense appears to be an on-going cat-and-mouse-game, similar to root detection.

Obfuscating keys in memory. While relying solely on obscurity for security lacks robustness [CWE-656], however, in the case where content encryption keys must be inevitably loaded into the memory, one possibility is to make it harder for key scanning heuristics to identify secret keys in memory dumps.

Heuristics used in identifying memory regions of interests (*e.g.*, those that contain content fragments and cryptographic keys) typically assume their targets to occupy a contiguous region of memory [78, 88]. Additionally, they might also leverage the mathematical relation between the raw secret key and its derived key schedules to pinpoint the targets [88]. It remains to be seen whether obfuscations can be used to defeat these assumptions and make it more difficult for an attacker to recover secret keys from the memory.

Watermarking. An orthogonal line of protection is to use watermarking to make the origin of piracy traceable. Over the years, there are techniques developed to watermark multimedia like audios [106] and motion pictures [107], as well as textual contents [108, 109]. In some cases, however, attackers can remove trivially detectable watermarks. Resilience against detection and removal remains the main objectives of watermarking research.

Another weakness of relying on watermarking for piracy tracking is that detection often relies on the content being leaked and shared on the Internet, and it remains difficult to detect offline sharing and contents that are stolen but not shared at all.

Trusted Execution Environments. A potential game changer is the use of Trusted Execution Environments (TEEs), Since the traditional execution environment could potentially be under adversarial control, TEE vendors typically leverage separation mechanisms enforced by the hardware platform to create an isolated execution environment, the internal execution state of which not even the OS can inspect, though depending on implementations, cryptographic code running inside an isolated environment like Intel SGX might still be susceptible to cache timing attacks [65, 66].

Various TEE implementations have been made available in recent years, especially on mobile platforms, where it has been reported that there are multiple vendors offering various TEE solutions that do not seem to conform to the same API standard [79], which might have made it hard for developers to implement a one-size-fits-all solution that is universally deployable across brands of devices, potentially hindering adoption.

In the TEE trust model, the vendors would typically serve as the root of trust, and they often employ a tight admission control model which might require potentially costly licensing and non-disclosure agreements prior to app development and deployment. This could potentially create a market where only big companies can afford to compete in, shutting out small businesses and individual developers. For some, this also presents a concern for consumer rights. Since the trust on TEE vendors who enforce admission control on what can be executed in the isolated environment is *transitive* (users trust the vendor, which in turn trust the TEE licensees to provide opaque but non-malicious software), the lack of transparency makes it difficult to detect subtle attacks (*e.g.*, spying and tracking) and hold the vendors accountable. While for cloud service providers, the ability to create and attest isolated execution environments might add appeals to their customers, it remains unclear how, if given the choice, consumers would be willing to pay for a hardware technology that they cannot control and cannot opt-out, instead of choosing the low cost devices without these hassles that are more customizable and configurable.

3.6 Conclusion

In this research, we shed light on the current practices and weaknesses of content delivery apps on mobile platforms, with concrete attacks on 141 apps. Due to some unjustified trust assumptions and weak design patterns, given the right adversary capability, it is often possible to bypass the content protection mechanism in place to achieve unrestricted access to raw contents. Feasibility of such attacks might have contributed to the conventional printed media's struggle for revenue. Content owners should evaluate the robustness of the app design before retaining the service of a developer.

Our findings present an interesting dilemma for content distributors to consider: either risk losing controls over contents by allowing untrustworthy devices to access their services, or risk losing customer reach. We hope that our work would bring awareness to the situation, and spark further research on identifying more app weaknesses. In particular, with more sophisticated frameworks like [110] and recent advancements in transparent debugging against anti-debugging and anti-instrumentation techniques [104, 105] we expect more apps can be reverse engineered and analyzed.

A Call to Arms

Another goal of this work is to summarize weakness patterns so that future developments of similar apps can benefit from the insights provided by this research, take various attack strategies into consideration, and avoid similar pitfalls.

Penetration testing becomes especially important in the case when content distributors are unwilling to completely shut off their services to customers who own only low-end devices without TEE capabilities, making obfuscation the only feasible partial solution. In the absence of a generic framework for quantifying the complexity of obfuscation, penetration testing becomes perhaps the only way to empirically evaluate how difficult it would be to extract secret states. Companies who already have an in-house red team could perhaps leverage it for this purpose.

3.7 Table of Apps and CWEs

The complete list of content delivery apps that we studied can be found in Table 3.1. The list of CWEs discussed in this research can be found in Table 3.2.

Table 3.1.: List of content distribution apps studied in this research

App Name	Version	Publisher	† Latest In-App Issue Cover Date	Latest In-App Issue Price ($)	‡ Category	‡ Installs	Attacks and Adversaries Discussed ❖
Group-1 Apps (Vendor: Apazine)							
The MagPi	5.0.3	The Raspberry Pi Foundation	Apr, 2018	3.99	News & Magazines	50000+	EVA (Sect. 3.3.1, $\mathcal{A}_{\text{Net(Sniff)}}$) PBA (Sect. 3.3.2, $\mathcal{A}_{\text{Net(Sniff)}}$) PBA (Sect. 3.4.1, $\mathcal{A}_{\text{ExS(R)}}$) KEA (Sect. 3.4.3, $\mathcal{A}_{\text{ExS(R)}}$)
Business Money	5.0	Business Money Promotions	Mar, 2018	14.99	News & Magazines	1000+	
Artists & Illustrators	5.1	The Chelsea Magazine Company	Jun, 2018	5.99	Lifestyle	1000+	
My MS-UK	5.0.3	MS-UK	Jan/Feb 2018	3.99	News & Magazines	100+	
Popshot Magazine	5.2	The Chelsea Magazine Company	Spring, 2018	7.49	Lifestyle	50+	
Group-2 Apps (Vendor: Apazine)							
Counter Intelligence Plus	4.0.0	Communications International Group	Jan 05, 2017	Free	Medical	1,000+	KEA (Sect. 3.4.5, $\mathcal{A}_{\text{InS(R)}}$)
Group-3 Apps (Vendor: Maz Systems)							
Forbes Magazine Ψ	6.1.0	Forbes Media	May 31, 2018	5.99	Business	100,000+	PBA (Sect. 3.3.2, $\mathcal{A}_{\text{Net(Sniff)}}$) EVA (Sect. 3.3.3, $\mathcal{A}_{\text{Net(TLSInt)}}$) Privacy Threats (Sect. 3.3.5) CEA (Sect. 3.4.4, $\mathcal{A}_{\text{InS(R)}}$) PBA (Sect. 3.4.7, $\mathcal{A}_{\text{InS(R+W)}}$)
Harvard Business Review	4.6.15 4.6.56 #	Harvard Business Publishing	May 01, 2018	18.99	Business	10,000+	
Designs in Machine Embroidery	6.1.0	Designs in Machine Embroidery	May 01, 2018	4.99	Lifestyle	10,000+	
Diabetes Self-Management	6.1.0	Madavor Media	Jun 01, 2018	5.99	Health & Fitness	10,000+	
ForbesLife	6.1.0	Forbes Media	Nov 23, 2015	6.99	Lifestyle	10,000+	
Mother Earth News	6.1.0, 6.1.56 #	Ogden Publications, Inc.	Feb 01, 2018	5.99	Lifestyle	10,000+	
Boys' Life	6.1.4, 6.1.56 #	Boy Scouts of America	Jun 01, 2018	4.99	News & Magazines, Education	5,000+	
Craft Beer & Brewing Magazine	6.1.50	Unfiltered Media Group	May 09, 2018	9.99	Lifestyle	5,000+	
GRIT Magazine	6.1.0	Ogden Publications, Inc.	May 01, 2018	4.99	Lifestyle	5,000+	
Guitar World	6.1.15	NewBay Media	Jul 01, 2018	7.99	Music & Audio	5,000+	
Inside Lacrosse	6.1.0	American City Business Journals	May 01, 2018	4.99	Sports	5,000+	
Mother Earth Living	6.1.0	Ogden Publications, Inc.	May 01, 2018	5.99	Lifestyle	5,000+	
USA Today Sports Weekly	6.1.0, 6.1.59 #	Gannett Company	May 29, 2018	2.99	Sports	5,000+	
ABA Journal Magazine	6.1.0	American Bar Association	May 01, 2018	6.99	Business	1,000+	

American Cheerleader Magazine	6.1.0	Varsity Spirit	Mar 21, 2018	3.99	Sports	1,000+
BirdWatching	6.1.0	Madavor Media	May 01, 2018	5.99	Lifestyle	1,000+
BUST Magazine	6.1.0	Debbie Stoller and Laurie Henzel	Apr 01, 2018	4.99	Entertainment	1,000+
Cake Central Magazine	6.1.0	Cake Central Media Corp.	Apr 01, 2017	5.99	Lifestyle	1,000+
Scouting magazine	6.1.0	Boy Scouts of America	May 01, 2018	3.99	Lifestyle	1,000+
Faerie Magazine	6.1.0	Faerie Magazine	Mar 15, 2018	4.99	Lifestyle	1,000+
Grassroots Motorsports Mag	6.1.15	Tim Suddard	Jun 01, 2018	5.99	Lifestyle	1,000+
Guitar Player Magazine++	6.1.15	NewBay Media	Jun 01, 2018	6.99	Music & Audio	1,000+
JazzTimes	6.1.0	Madavor Media	Jun 01, 2018	3.99	Music & Audio	1,000+
Joy of Kosher Magazine	6.1.0	Kosher Network International	Nov 24, 2017	3.99	Lifestyle	1,000+
Kayak Angler	6.1.15	Rapid Media	Apr 01, 2018	3.99	Sports	1,000+
Leatherneck Magazine	6.1.0	Marine Corps Association	Jun 01, 2018	4.99	News & Magazines	1,000+
Marine Corps Gazette	6.1.0	Marine Corps Association	Jun 01, 2018	4.99	News & Magazines	1,000+
Motorcycle Classics Magazine	6.1.0	Ogden Publications, Inc.	May 01, 2018	6.99	Entertainment	1,000+
National Wildlife magazine	6.1.0	National Wildlife Federation	Apr 01, 2018	3.99	Education	1,000+
New York Observer	6.1.0	Observer Media	Nov 14, 2016	1.99	News & Magazines	1,000+
Paddling Mag	6.1.15	Rapid Media	Apr 01, 2018	2.99	Sports	1,000+
Paleo Magazine	6.1.12	Paleo Magazine	Jul 05, 2018	2.99	Health & Fitness	1,000+
The Writer	6.1.0	Madavor Media	Jul 01, 2018	5.99	Education	1,000+
V Magazine	6.1.0	Visionaire	May 03, 2018	3.99	Entertainment	1,000+
Volleyball Magazine	6.1.0	Volleyball World Wide	Apr 29, 2016	3.99	Sports	1,000+
Adventure Kayak+ Magazine	6.1.15	Rapid Media	Jun 01, 2017	3.99	Sports	500+
Art Photo Feature	6.1.0	APF Magazine	Feb 01, 2016	3.99	Photography	500+
City Journal	6.1.0	Manhattan Institute for Policy Research	Apr 15, 2018	3.99	Business	500+
Farm Collector	6.1.0	Ogden Publications, Inc.	Jun 01, 2018	4.99	Lifestyle	500+
Gluten-Free Living	6.1.0	Madavor Media	May 01, 2018	6.99	Health & Fitness	500+
Keyboard Magazine	6.1.0	NewBay Media	Jun 01, 2018	5.99	Music & Audio	500+

PBA (Sect. 3.3.2, $\mathcal{A}_{\mathsf{Net(Sniff)}}$)
EVA (Sect. 3.3.3, $\mathcal{A}_{\mathsf{Net(TLSInt)}}$)
Privacy Threats (Sect. 3.3.5)
CEA (Sect. 3.4.4, $\mathcal{A}_{\mathsf{InS(R)}}$)
PBA (Sect. 3.4.7, $\mathcal{A}_{\mathsf{InS(R+W)}}$)

Man of the World Magazine	6.1.0	Man of the World	Oct 17, 2016	9.99	Lifestyle	500+	
The Real Deal Magazine	6.1.0	Korangy Publishing	May 01, 2018	2.99	Business	500+	
Revolver Magazine	6.1.15	NewBay Media	Apr 01, 2018	6.99	Music & Audio	500+	
Art Business News	6.1.0	Redwood Media Group	Apr 01, 2016	0.99	Business	100+	
Animania Magazine	6.1.15	RSPCA NSW	Mar 01, 2018	3.99	Lifestyle	100+	
Pain-Free Living	6.1.0	Madavor Media	Jun 01, 2018	4.99	Health & Fitness	100+	
Bass Player+	6.1.15	NewBay Media	Jun 01, 2018	5.99	Music & Audio	100+	
Digital Video	6.1.15	NewBay Media	Jan 01, 2018	4.99	Productivity	100+	
Electronic Musician+	6.1.0	NewBay Media	Jun 01, 2018	5.99	Music & Audio	100+	
Guitar Aficionado	6.1.15	NewBay Media	Jan 01, 2018	7.99	Lifestyle	100+	
Hail Varsity Magazine	6.1.0	Hail Varsity	Feb 13, 2018	2.99	Sports	100+	
Inside Pitch	6.1.0	The American Baseball Coaches Association	May 01, 2018	1.99	Sports	100+	PBA (Sect. 3.3.2, $\mathcal{A}_{\mathsf{Net(Sniff)}}$) EVA (Sect. 3.3.3, $\mathcal{A}_{\mathsf{Net(TLSInt)}}$) Privacy Threats (Sect. 3.3.5) CEA (Sect. 3.4.4, $\mathcal{A}_{\mathsf{InS(R)}}$) PBA (Sect. 3.4.7, $\mathcal{A}_{\mathsf{InS(R+W)}}$)
Inside Weddings	6.1.0	Inside Weddings	Mar 13, 2018	5.99	Lifestyle	100+	
Multichannel News++	6.1.15	NewBay Media	May 14, 2018	6.99	Music & Audio	100+	
Mix Magazine+	6.1.0	NewBay Media	May 01, 2018	6.99	Music & Audio	100+	
MUSE Magazine	6.1.0	MUSE Magazine	Feb 19, 2018	3.99	Lifestyle	100+	
National Affairs	6.1.15	National Affairs, Inc.	Mar 21, 2018	3.99	Education	100+	
TWICE+	6.1.0, 6.1.56 #	NewBay Media	May 21, 2018	9.99	Music & Audio	100+	
AV Technology	6.1.12	NewBay Media	May 01, 2018	5.99	Business	50+	
Broadcasting & Cable++	6.1.8	NewBay Media	May 21, 2018	6.99	Music & Audio	50+	
Deli Business	6.1.0	Phoenix Media Network	Dec 01, 2017	14.99	Business	50+	
HeirlmGardnrMag	6.1.15	Ogden Publications, Inc.	Mar 01, 2018	9.99	Lifestyle	50+	
Sound Video Contractor	6.1.15	NewBay Media	May 01, 2018	6.99	Business	50+	
Digital Signage	6.1.15	NewBay Media	Apr 27, 2018	5.99	Business	10+	
Resident Sys	6.1.15	NewBay Media	Jun 01, 2018	4.99	Business	10+	
System Contractor News	6.1.15	NewBay Media	Jun 01, 2018	5.99	Business	10+	
Tech&Learning	6.1.15	NewBay Media	May 01, 2018	6.99	Education	10+	
Pro Sound News §	6.1.15	NewBay Media	May 01, 2018	5.99	Business (Amazon App Store)	Ranked 1250 in Business	
Revista La Fuente §	6.1.0	Revista La Fuente	Jun 01, 2018	2.99	Lifestyle (Amazon App Store)	Ranked 7367 in Lifestyle	

Group-4 Apps (Vendor: Maz Systems)							
Bloomberg Business-week+	2.4.6	Bloomberg L.P.	Subscription-based	59.99 yearly	News & Magazines	10,000,000+	PBA (Sect. 3.3.4, $\mathcal{A}_{\mathsf{Net(TLSInt)}}$) PBA (Sect. 3.4.6, $\mathcal{A}_{\mathsf{InS(R)}}$)
Salon.com	2	Salon Media Group	Subscription-based	49.99 yearly	News & Magazines	1000+	
Group-5 Apps (Vendor: Maz Systems)							
Entrepreneur Magazine	10020	Entrepreneur Media	Jun, 2018	4.99	Business	100,000+	PBA (Sect. 3.3.4, $\mathcal{A}_{\mathsf{Net(TLSInt)}}$) PBA (Sect. 3.4.6, $\mathcal{A}_{\mathsf{InS(R)}}$)
Forbes Magazine Ψ	10020	Forbes Media	Jun, 2018	5.99	Business	100,000+	
Diesel World	10010	Engaged Media	Aug, 2018	5.99	News & Magazines	10000+	
Knives Illustrated	10020	Engaged Media	Jul, 2018	4.99	News & Magazines	5000+	
Gun World	10020	Engaged Media	Jul, 2018	3.99	News & Magazines	5000+	
American Survival Guide	10020	Engaged Media	Jul, 2018	6.99	News & Magazines	5000+	
Ultimate Diesel Builders Guide	10020	Engaged Media	Jun, 2018	5.99	News & Magazines	5000+	
Outside TV Features	10021	Outside Television	Subscription-based	4.99 monthly	Sports	1,000+	
Inc. Must Reads and Magazine	10020	Mansueto Ventures	Jun, 2018	4.99	Business	1,000+	
The Nation Magazine	10020	The Nation Company	May 28, 2018	1.99	News & Magazines	1,000+	
Conceal & Carry	10020	Engaged Media	Summer, 2018	7.99	News & Magazines	1000+	
Cottages & Bungalow	10020	Engaged Media	Jun, 2018	8.99	News & Magazines	1000+	
The Rebel Media	10020	The Rebel News Network	Subscription-based	84.99 yearly	News & Magazines	1,000+	
Lion's Roar Magazine	10020	Lion's Roar Foundation	Subscription-based	23.99 yearly	Lifestyle	500+	
Buddhadharma	10020	Lion's Roar Foundation	Subscription-based	23.99 yearly	News & Magazines	500+	
Texas Monthly	10020	Texas Monthly	Jun, 2018	4.99	News & Magazines	500+	
All About Beer	10020	All About Beer	Mar, 2018	4.99	Food & Drink	100+	
Atomic Ranch	10020	Engaged Media	Summer, 2018	5.99	News & Magazines	100+	
FNF Coaches	10020	A.E. Engine	Apr, 2018	varies	Sports	100+	
Tread Magazine	10020	Engaged Media	May, 2018	7.99	Lifestyle	100+	
Vogue Knitting	1	Soho Publishing	Spring, 2018	5.99	News & Magazines	100+	
American Farmhouse Style	10020	Engaged Media	Summer, 2018	7.99	Lifestyle	10+	
Berko	10020	Pat Callinan Media	May 10, 2018	5.99	Lifestyle	10+	

Group-6 Apps (Vendor: Maz Systems)							
Men's Health Magazine	10020	Hearst Communications	Jun, 2018	4.99	Health & Fitness	10,000+	PBA (Sect. 3.3.4, $\mathcal{A}_{\mathsf{Net(TLSInt)}}$) CEA (Sect. 3.4.2, $\mathcal{A}_{\mathsf{ExS(R)}}$)
Runner's World	10020	Hearst Communications	Jul, 2018	4.99	Health & Fitness	1,000+	
Women's Health Mag	10020	Hearst Communications	Jun, 2018	4.99	Health & Fitness	1,000+	
Prevention	10020	Hearst Communications	Jun, 2018	4.99	Health & Fitness	500+	
Quilting	1	F+W Media	Jun, 2018	6.99	Education	500+	
Bicycling	10020	Hearst Communications	Jul, 2018	4.99	Health & Fitness	100+	
FNF — Friday Night Football	10020	A.E. Engine	2017	0.99	Sports	100+	
IW Knits	1	F+W Media	Summer, 2018	7.99	Education	50+	
ArtistsMag	1	F+W Media	Jul, 2018	6.99	Education	50+	
Group-7 Apps (Vendor: Pugpig)							
The Independent Daily Edition	4.5.1313.370	Independent Digital News & Media Limited	Wednesday 27 Jun, 2018	168.74 yearly	News & Magazines	50,000+	PBA (Sect. 3.3.4, $\mathcal{A}_{\mathsf{Net(TLSInt)}}$)
Primal 9	1.1.3804.716	Hearst Magazines UK	2018	49.99	Health & Fitness	10,000+	
Cosmopolitan UK	6.5.1655.377	Hearst Magazines UK	Aug, 2018	9.99 yearly	Lifestyle	5,000+	
Glamour Magazine (UK)	1.7.2137.893	The Condé Nast Publications Limited	Spring/Summer, 2018	0.99	News & Magazines	5,000+	
Wired UK	33.3.187.893	The Condé Nast Publications Limited	May/Jun, 2018	17.99 yearly	News & Magazines	5,000+	
Condé Nast Traveler Magazine	1.2.1189.893	The Condé Nast Publications Limited	Jun, 2018	29.99 yearly	Lifestyle	1,000+	
GQ Style UK	1.2.3455.893	The Condé Nast Publications Limited	Spring/Summer, 2018	9.99 yearly	Lifestyle	1,000+	
Tatler	1.2.1189.893	The Condé Nast Publications Limited	Jul, 2018	29.99 yearly	Lifestyle	1,000+	
WH Transform	1.0.2982.490	Hearst Magazines UK	2018	54.99	Health & Fitness	1,000+	
Brides	1.2.1189.893	The Condé Nast Publications Limited	May/Jun, 2018	23.99 yearly	Lifestyle	500+	
House & Garden	1.2.1189.893	The Condé Nast Publications Limited	Jul, 2018	30.99 yearly	Lifestyle	500+	
Reveal UK	6.5.31.50.377	Hearst Magazines UK	Week 26, 2018	26.99 yearly	Entertainment	100+	
The World of Interiors	1.1.326.893	The Condé Nast Publications Limited	Jul, 2018	35.99 yearly	Lifestyle	100+	
QP Magazine	1.0.3390.1350	Hearst Magazines UK	Jun, 2018	12.99 yearly	News & Magazines	10+	

Group-8 Apps (Vendor: Pugpig)							
ELLE Magazine UK	6.5.1655.377	Hearst Magazines UK	Jun, 2018	29.99 yearly	Lifestyle	5,000+	PBA (Sect. 3.3.4, $\mathcal{A}_{\mathsf{Net(TLSInt)}}$) CEA (Sect. 3.4.4, $\mathcal{A}_{\mathsf{InS(R)}}$)
Men's Health UK	1.1.3804.716	Hearst Magazines UK	Jul, 2018	29.99 yearly	Lifestyle	5,000+	
ELLE Decoration UK	6.5.1655.377	Hearst Magazines UK	Jul, 2018	34.99 yearly	Lifestyle	1,000+	
Esquire UK	6.5.1665.377	Hearst Magazines UK	Jul/Aug, 2018	19.99 yearly	Lifestyle	1,000+	
Good House-keeping UK	6.5.1655.377	Hearst Magazines UK	Jun, 2018	29.99 yearly	Lifestyle	1,000+	
Harper's Bazaar UK	6.5.1655.377	Hearst Magazines UK	Jul, 2018	35.99 yearly	Lifestyle	1,000+	
Inside Soap UK	6.5.1655.377	Hearst Magazines UK	Week 26, 2018	59.99 yearly	Entertainment	1,000+	
Runner's World UK	6.5.1655.377	Hearst Magazines UK	Aug, 2018	35.99 yearly	Health & Fitness	1,000+	
Women's Health UK	1.5.3696.377	Hearst Magazines UK	Jul, 2018	29.99 yearly	Lifestyle	1,000+	
House Beautiful UK	6.5.1655.377	Hearst Magazines UK	Aug, 2018	30.99 yearly	Lifestyle	500+	
Country Living UK	6.5.1655.377	Hearst Magazines UK	Jun, 2018	29.99 yearly	Lifestyle	100+	
Prima UK	6.5.1655.377	Hearst Magazines UK	Jul, 2018	28.99 yearly	Lifestyle	100+	
Real People UK	6.5.1655.377	Hearst Magazines UK	Week 27, 2018	26.99 yearly	Entertainment	100+	
Red Magazine UK	6.5.1655.377	Hearst Magazines UK	Jul, 2018	23.99 yearly	Lifestyle	100+	
Town & Country	6.5.1655.377	Hearst Magazines UK	Summer, 2018	18.99 yearly	Lifestyle	100+	
Best UK	6.5.1655.377	Hearst Magazines UK	Week 26, 2018	30.99 yearly	Entertainment	50+	
Group-9 Apps (Vendor: Amazon)							
Amazon Music	6.5.3	various	2-tier subscriptions		Music & Audio	100,000,000+	KEA (Sect. 3.4.8, $\mathcal{A}_{\mathsf{Mem+BinIns}}$)
Audible	2.25.0	various	various subscription plans exist		Books & Reference	100,000,000+	

§ These Apps were only available on Amazon App Store but not on Google Play Store.

‡ Information in these columns retrieved from Google Play Store in Jun, 2018.

# For these apps, the newer version uses an encrypted database. They are however still susceptible to PBAs through attack surface **(AS1)** as discussed in Section 3.3.2.

Ψ The new version was released during the course of our study to replace the old one. The varied designs are susceptible to different attacks, though the pricing stays the same.

† Some cover dates are in the future due to 1) some have long intervals between issues; 2) some offer digital access earlier than in print. ($) All prices are in US dollars.

❖ KEA = Key Extraction Attacks; PBA = Purchase Bypass Attacks; CEA = Content Extraction Attacks;
EVA = Eavesdropping Attacks. In this research, KEA and EVA both imply CEA.

Table 3.2.: List of CWEs discussed in this research

CWE ID	Name	Description
CWE-313	Cleartext Storage in a File or on Disk	The application stores sensitive information in cleartext in a file, or on disk.
CWE-319	Cleartext Transmission of Sensitive Information	The software transmits sensitive or security-critical data in cleartext in a communication channel that can be sniffed by unauthorized actors.
CWE-354	Improper Validation of Integrity Check Value	The software does not validate or incorrectly validates the integrity check values or "checksums" of a message. This may prevent it from detecting if the data has been modified or corrupted in transmission.
CWE-425	Direct Request ('Forced Browsing')	The web application does not adequately enforce appropriate authorization on all restricted URLs, scripts, or files.
CWE-454	External Initialization of Trusted Variables or Data Stores	The software initializes critical internal variables or data stores using inputs that can be modified by untrusted actors.
CWE-532	Information Exposure Through Log Files	Information written to log files can be of a sensitive nature and give valuable guidance to an attacker or expose sensitive user information.
CWE-603	Use of Client-Side Authentication	A client/server product performs authentication within client code but not in server code, allowing server-side authentication to be bypassed via a modified client that omits the authentication check.
CWE-642	External Control of Critical State Data	The software stores security-critical state information about its users, or the software itself, in a location that is accessible to unauthorized actors.
CWE-654	Reliance on a Single Factor in a Security Decision	A protection mechanism relies exclusively, or to a large extent, on the evaluation of a single condition or the integrity of a single object or entity in order to make a decision about granting access to restricted resources or functionality.
CWE-656	Reliance on Security Through Obscurity	The software uses a protection mechanism whose strength depends heavily on its obscurity, such that knowledge of its algorithms or key data is sufficient to defeat the mechanism.
CWE-921	Storage of Sensitive Data in a Mechanism without Access Control	The software stores sensitive information in a file system or device that does not have built-in access control.

4. EXPOSING NONCOMPLIANCE IN IMPLEMENTATIONS OF X.509 CERTIFICATE VALIDATION WITH A PRINCIPLED APPROACH

4.1 Introduction

The X.509 Public-Key Infrastructure (PKI) standard [111,112] has long been used in SSL/TLS as a means to distribute keys and provide authentication. The security assurance expected from SSL/TLS handshake critically hinges on the premise that communication peers, particularly the clients, correctly perform the prescribed validation of the server-provided X.509 certificate chain. Put differently, *correctly validating X.509 certificate chains is imperative to achieving security.* Flaws in implementations of the certificate chain validation logic (**CCVL**) could potentially lead to two pitfalls: (1) Overly restrictive CCVL (*i.e.*, incorrectly rejecting valid certificate chains) may result in **interoperability issues** and potential loss of service; (2) Overly permissive CCVL (*i.e.*, incorrectly accepting invalid certificate chains) may allow attackers to conduct **impersonation attacks**. We call an X.509 CCVL implementation **non-compliant with the X.509 specification** if it suffers from over-permissiveness, over-restrictiveness, or both. The X.509 standard [111] is defined in a generic way to accommodate different usage scenarios (*e.g.* for code signing, encipherment, authentication, etc.). In this work, we concentrate on X.509's use in the context of Internet communication (*i.e.*, clients performing server authentication during SSL/TLS negotiation) and focus on the RFC 5280 specification [112].

Although the SSL/TLS protocol implementations have undergone extensive scrutiny [21–23, 55–57], similar rigorous investigation is absent for checking compliance of X.509 CCVL implementations. For instance, researchers have developed a formally verified reference implementation for the SSL/TLS protocol [22] but it does not include a formally verified CCVL. The portion of code in SSL/TLS libraries responsible for performing the X.509 chain validation are often plagued with severe bugs [113–125].

Implementing a compliant X.509 CCVL is not easy, primarily due to the complexity of its requirements. For example, through our analysis, we have seen how a supposedly simple boundary check on date and time can lead to various instances of noncompliance in different libraries due to mishandling time zones and misinterpreting the specification. The following comment from an SSL/TLS library developer that we contacted regarding a bug report concisely capture the intricacy of the task: "*In general, X.509 validation is one of the most error prone, code bloating, and compatibility nightmares in TLS implementation.*"

There are two possible directions for addressing X.509 CCVL's noncompliance problem: (1) Formally proving compliance of a (possibly reference) CCVL implementation with respect to the specification and having every library use it; (2) Devising approaches for finding noncompliance in CCVL implementations. The difficulty of automatically proving compliance of an X.509 CCVL implementation, in addition to the problem being undecidable in general [126], stems from the fact that standard formal verification techniques [127–146] often do not support all the idiosyncrasies of a system level programming language like C. The direction of finding noncompliance was adopted by Brubaker et al. [1] and they uncovered a number of bugs in the CCVL implementations using black-box fuzzing, which raised awareness on both the existence and severity of the problem. Our approach is also geared towards finding noncompliance in real CCVL implementations.

Although black-box fuzzing is an effective technique for finding implementation flaws, especially when the source code is not available, it suffers from the following well known limitation: given a vast input space, black box fuzzing fails to concentrate on relevant portions of the source code without explicit guidance (*i.e.* lack of code coverage).

Symbolic execution [147] has been found to address the above limitation [35–37]. Symbolic execution is also known to be effective in finding bugs buried deep in the execution. It is, however, cursed by the problem of *path explosion* [148], which severely hinders its scalability and practicality, especially when the input is recursively structured and complex as in the case of X.509 certificates.

In this research, we take the first step in making symbolic execution practical for finding noncompliance in real X.509 implementations. To this end, we solve symbolic execution's path explosion problem in the following manner: (1) Focusing our analysis on open source SSL/TLS libraries that have a small footprint and code base; (2) Applying a combination of domain-specific insights, abstractions, and compartmentalization techniques to the symbolic execution environment.

Small footprint SSL/TLS libraries are typically tailor-made for resource constrained platforms, and often prioritize efficiency over robustness. With the emergence of Internet-of-Things (IoT), these libraries are actively deployed on commodity devices to satisfy the needs for secure communication in the IoT ecosystem [149–152]. Furthermore, following the discovery of several high-profile vulnerabilities due to implementation flaws in recent years [153–155], traditional SSL/TLS libraries like OpenSSL has been criticized to have an unnecessarily large and messy code base that is both slow and infested with bugs [156]. A call for diverse alternative implementations with better maintainability and a desire for performance have sparked interests in adopting small footprint SSL/TLS libraries for building applications on even conventional PC platforms [157–161]. Hence it is of interest for us to evaluate these implementations of X.509 validation for robustness and compliance to specification.

To make symbolic execution practical and feasible, we develop the concept of **SymCerts**, which are syntactically well-formed symbolic X.509 certificate chains, such that each certificate contains a mix of concrete and symbolic values. To further reduce path explosion, we decompose the problem of noncompliance finding into smaller independent sub-problems based on the domain-specific observation that **some** certificate fields are logically independent in their semantic meanings. Fields in the same sub-problem are made symbolic at the same time, while the other unrelated fields are kept concrete. The use of SymCerts, along with the observation of semantic independence of fields, address the path explosion problem of symbolic execution that stem from the recursive and complex nature of the input certificate chain representation.

Approach: An X.509 CCVL *partitions* the certificate chain input universe into accepting (chains deemed valid) and rejecting (chains deemed invalid) certificate universes. We use symbolic execution to automatically extract the approximation of the certificate accepting and rejecting universes (See Figure 4.1), and symbolically represent these sets as path constraints (quantifier-free first order logic formulas), where the symbolic variables correspond to fields and extensions of certificates.

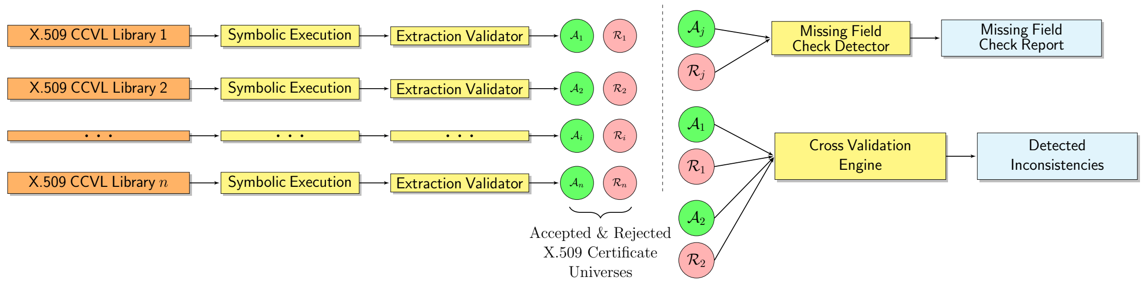

X.509 CCVL Library 1
X.509 CCVL Library 2
. . .
X.509 CCVL Library n
Symbolic Execution
Extraction Validator
$\mathcal{A}_1$ $\mathcal{R}_1$
$\mathcal{A}_2$ $\mathcal{R}_2$
$\mathcal{A}_i$ $\mathcal{R}_i$
$\mathcal{A}_n$ $\mathcal{R}_n$
Accepted & Rejected X.509 Certificate Universes
$\mathcal{A}_j$ $\mathcal{R}_j$
Missing Field Check Detector
Missing Field Check Report
Cross Validation Engine
Detected Inconsistencies

Fig. 4.1.: Our noncompliance finding approach for X.509 CCVL implementations

In our approach, **symbolic execution engine** takes as input a CCVL implementation and extracts the approximated accepted and rejecting certificate universe whereas **extraction validator** validates it through concrete execution. **Missing field check detector** finds unscrutinized certificate fields from the universes. **Cross validation engine** performs cross validation among two implementations universes.

In the case where an X.509 CCVL implementation is noncompliant due to the lack of certain checks, a simple search (*e.g.* with `grep`) of the path constraints will uncover such noncompliance, as the corresponding symbolic variables will not appear in the extracted path constraints. For catching deeper noncompliance, we leverage the principle of *differential testing* [58, 162], by carrying out a cross validation of different implementations. Given two implementations I_1, I_2, and their corresponding accepting and rejecting certificate universes $\mathcal{A}_1$, $\mathcal{R}_1$, $\mathcal{A}_2$, and $\mathcal{R}_2$, we can automatically determine whether discrepancies exist between I_1 and I_2 (*i.e.*, one implementation accepts a certificate chain whereas the other rejects it) by checking whether the sets $\mathcal{A}_1 \cap \mathcal{R}_2$ and $\mathcal{A}_2 \cap \mathcal{R}_1$ are nonempty. Representing these sets symbolically enables us to implement the set intersection operator by leveraging a Satisfiability Modulo Theory (SMT) solver [163, 164].

Evaluation and Findings: We analyzed 9 implementations from 4 families of code base (axTLS, wolfSSL, mbedTLS, MatrixSSL) and uncovered 48 instances of noncompliance.

Notably, we have detected the erroneous logic embraced by wolfSSL 3.6.6 and MatrixSSL 3.7.2 for matching ExtKeyUsage object identifiers (OID); such OID matching is used to assert the proper use of the key according to its intended purposes (*e.g.*, for code signing). Although standard usage purposes are identified with pre-defined values (*e.g.*, 1.3.6.1.5.5.7.3.1 means server authentication), other values are allowed for defining custom purposes. Both wolfSSL 3.6.6 and MatrixSSL 3.7.2 take a summation of the encoded bytes of an OID, and uses only the sum for matching against known standard key usage purposes. In their scheme, OID 1.3.6.1.5.5.7.3.1 (ASN.1 DER-encoded bytes: 0x2B 0x06 0x01 0x05 0x05 0x07 0x03 0x01) will be identified as decimal 71. Despite OIDs being unique hierarchically, the summation of their encoded bytes may not be. An adversary may request a certificate authority to issue an innocuous-looking certificate with a custom key usage purpose OID value that adds up to 71, and would then be able to use it for server authentication in these libraries. We have reported this bug to the library developers. They **acknowledged the problem and have it fixed** in new releases.

Another notable finding is the misinterpretation of the year field of UTCTime by MatrixSSL 3.7.2. In UTCTime format, the RFC prescribes two bytes YY to denote years such that YY$\in [0, 49]$ is treated as the year 20YY whereas YY$\in [50, 99]$ is treated as 19YY, allowing years to be in range $1950 - 2049$. However, MatrixSSL 3.7.2 misinterprets the YY field and hence miscalculates some certificate **expiration by** 100

years (*e.g.*, certificates expired in 1995 are considered to expire in 2095). Developers of MatrixSSL **acknowledged this bug** after receiving our report and **implemented a fix** in a newer version. Other findings are reported in 4.5.

Contributions: In summary, this research makes the following contributions:

1. We take the first step towards developing a more principled approach to systematically analyze real implementations of X.509 validation.
2. Though scalability issue exists, we show that symbolic execution could be made practical by limiting the scope of analysis and using domain specific optimization, and it is very effective in exposing implementation flaws.
3. We revisit three specific implementations that have been studied before in the literature [1]. With new findings that are otherwise difficult to find with an unguided fuzzing approach, we show that previous work based on fuzz testing indeed suffers from false negatives, and some of their claims are inaccurate due to a possible misinterpretation of those false negatives.
4. For the other and more recent implementations that had not been studied before, we found multiple instances of noncompliance and have them reported to the developers.

4.2 Background and Problem Definition

In this section, we first present a brief introduction on X.509 certificates and their validation logic. We then present the noncompliance finding problem and the associated high level challenges.

4.2.1 Preliminary on X.509 Certificate Validation

The X.509 PKI standard is described in ITU-T Recommendation X.509 [111]. The certificate format itself, at the time of writing, has 3 versions. Version 2 and

3 were introduced to add support for unique identifiers and certificate extensions, respectively. X.509 certificates can be used in various environments for different purposes. A variety of standard certificate extensions are defined in the standard documents [111] and ANSI X9.55. RFC 5280 [112] profiles how version 3 certificates, extensions and CRLs are meant to be used specifically for the Internet. Since we focus on this particular prominent use case of X.509, in the rest of this section, we provide a simplified overview of what makes a certificate and how validation should happen in general, taking the viewpoint of an Internet client and using RFC 5280 as the main reference.

Contents of an X.509 certificate

At a very high level, a X.509 certificate is made of 3 parts: the TBS (To-Be-Signed) part, which includes most of the semantic content of the certificate; a signature algorithm identifier, which denotes the algorithm the issuer used to sign the certificate; and finally the actual signature value. The TBS part generally includes the following fields: version (version number), serialNumber (that can uniquely identify a certificate), signature (the signature algorithm identifier), issuer (name of the entity who signed the certificate), validity (a time period of which the certificate can be considered as valid), subject (name of the subject of the certificate), subjectPublicKeyInfo (the public key of the subject of the certificate).

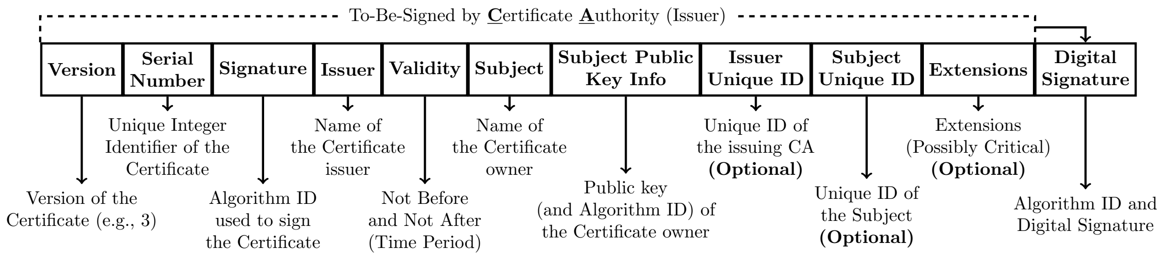

To-Be-Signed by Certificate Authority (Issuer)
Version
Serial Number
Signature
Issuer
Validity
Subject
Subject Public Key Info
Issuer Unique ID
Subject Unique ID
Extensions
Digital Signature
Unique Integer Identifier of the Certificate
Name of the Certificate issuer
Name of the Certificate owner
Unique ID of the issuing CA (Optional)
Extensions (Possibly Critical) (Optional)
Version of the Certificate (e.g., 3)
Algorithm ID used to sign the Certificate
Not Before and Not After (Time Period)
Public key (and Algorithm ID) of the Certificate owner
Unique ID of the Subject (Optional)
Algorithm ID and Digital Signature

Fig. 4.2.: A simplified structural view of an X.509 version 3 certificate

This drawing is inspired by a similar figure in [165].

Towards the end of the TBS of a X.509 version 3 certificate there are three optional constructs: the issuerUniqueID and subjectUniqueID, which are respectively unique identifiers of the issuer and subject of the certificate, followed by extensions, which is a sequence of X.509 version 3 extensions. See Figure 4.2 for a simplified visualization of the structure of a typical X.509 version 3 certificate.

X.509 certificate validation

The X.509 PKI is based on the idea of "chain of trust". The main objective of certificate validation is to show that given a trust anchor, C_0, the trust can be extended through a chain of certificates, all the way down to the communication peer (*e.g.* a specific server). Hence the basic check requires that for each certificate of a chain, the issuer name of a certificate C_i must equal to the subject name of the previous certificate C_{i-1}, and the signature on C_i can be correctly verified using the algorithm, the public key and other parameters derived from C_{i-1}.

In addition, each certificate involved in forming the chain of trust must be currently valid, in the sense that the current system time should be within the range (inclusively) prescribed by the notBefore and notAfter attributes of the Validity field.

Other checks in X.509 certificate validation are related to the handling of version 3 extensions. Extensions give CAs a means to impose additional restrictions on certificates issued by them, to avoid abuse of certificates.

Extensions can be marked as critical or non-critical. For the standard set of extensions, RFC 5280 [112] mandates some default criticality that a conforming CAs should follow. However, from the point of view of a certificate-using system, extensions should be processed regardless of their criticality if the system is able to, and in case it cannot process any of the critical extensions then the certificate should be rejected.

On a valid certificate chain, each of the certificates needs to be a CA certificate, except for the leaf one (both CA and non-CA are allowed). In X.509 version 3, this is achieved by checking the basicConstraints extension, which contains an isCA boolean field indicating whether the certificate is a CA certificate or not, and an optional integer pathLenConstraint that limits the number of non-self-issued intermediate CA certificates that can follow on the chain, not counting the leaf one. Before version 3, X.509 certificates do not have extensions. In such cases, clients can choose to either consider those to be non-CA certificates, or use an out-of-band mechanism to verify if those are CA certificates or not.

The KeyUsage and ExtKeyUsage are two useful extensions that describe the intended purposes of a certificate. With issuing CAs imposing these on certificates, and clients faithfully checking the intended purposes, some certificate abuse scenarios can be stopped (*e.g.* using a certificate that is only issued for signing software in a SSL/TLS handshake for authentication would not be allowed).

There are other standard extensions which we do not present here. For a complete list of extensions deemed useful for the Internet, and the details on how to handle them, we refer the readers to RFC 5280 [112].

Sources of noncompliance

The intricacies of implementing a compliant X.509 CCVL stem from the rich set of fields in certificates, which are further complicated by their wide range of possible values, as well as the numerous optional but possibly critical extensions. Noncompliance can occur due to the following two reasons:

1. Certain fields and/or extensions that must be checked are not involved in the decision making procedure of a CCVL implementation. This can be further divided into:

   (a) The fields and/or extensions are not being parsed into an internal data structure. This is mostly due to a lack of intention to support a thorough and robust check, possibly due to concerns on resource usage.

   (b) The fields and/or extensions are being parsed into an internal data structure but checks did not happen. This is mostly due to an intention to perform the checks but the implementation is not complete.

2. The fields and/or extensions are involved into deciding whether to accept or reject the chain, but due to coding and/or logical errors in the parsing code and/or validation code, the checks are not performed correctly.

4.2.2 Goal and Challenges

In this research, our goal is to **check whether a given X.509 CCVL implementation is compliant with the X.509 specification**. There are two ways to go about checking compliance of an implementation, namely, (1) proving the compliance of the implementation with respect to the specification and (2) trying to find noncompliance in the implementation. Our approach is geared towards finding noncompliance.

Why Not Prove Compliance

To prove compliance of a given CCVL implementation, we have to formally specify the valid sets of X.509 certificate chains that a CCVL implementation should accept. The X.509 specification is, however, described in natural languages and coming up with a complete formal specification is cumbersome and error-prone. Furthermore, even if we have such a formal specification Ψ at our disposal, proving that Ψ is satisfied by the CCVL implementation I (*i.e.*, $I \models \Psi$) using standard formal verification techniques [127–146] is infeasible as the problem is undecidable in general [126]. Also, formal verification techniques often do not scale and support real implementations. For this reason, we resort to noncompliance finding in the implementation.

Challenges

We now discuss the inherent challenges of the noncompliance finding.

Natural Languages Specification The X.509 specification is written in English and it is inherently prone to under-specification, ambiguities, inconsistencies, and misinterpretations. To validate a noncompliant instance it is often required to consult the specification when we do not have a formal specification at our disposal. We resort to manual effort to address this challenge.

Scalability The complex format of X.509 certificates and also the intricacies in certificate chain validation make it difficult to develop a scalable noncompliance checker. Also, it is difficult to develop a scalable noncompliance checker for real libraries written in system level languages such as C.

Cryptographic Libraries A X.509 CCVL relies on cryptographic functions to perform operations such as digital signature verification. Cryptographic functions are well recognized to be difficult to automatically analyze for correctness.

4.3 Our Noncompliance Finding Approach

In this section, we first briefly describe symbolic execution and then present how we leverage it for noncompliance detection. Finally, we discuss several technical challenges of applying symbolic execution and how we overcome them.

4.3.1 Preliminary on Symbolic Execution

Symbolic execution [147] has been shown to be an effective way of inferring test cases that yield high code coverage [34–43].

It achieves this objective by running a program with symbolic values for input variables. During execution, when it encounters a branch instruction (*e.g.*, if-else) with a branching condition on symbolic values, it consults a Satisfiability Modulo Theory (SMT) solver [163,164] to check whether any of the two branches (*i.e.*, the if and else branches) are possible according to their branching conditions. If any of the branches are feasible (*i.e.*, the branching conditions are satisfiable for some concrete values for the input variables), the execution explores the corresponding paths. It keeps collecting all the feasible branching conditions on the input (symbolic) variables, also known as *path constraints*, until the program terminates or reaches a point of interest (*e.g.*, an error location). It then consults an SMT solver to obtain concrete values for the input that will induce the path in question.

4.3.2 Approximating Universes with Symbolic Execution

For noncompliance detection, our approach critically relies on extracting the universes of accepted and rejected certificate chains induced by a given X.509 CCVL implementation.

Suppose we denote the universe of all possible X.509 certificate chains with $\mathcal{C}$, a given X.509 CCVL partitions $\mathcal{C}$ into two sets $\mathcal{A}$ (the set of accepting certificate chains) and $\mathcal{R}$ (the set of rejecting certificate chains) such that $\mathcal{C} = \mathcal{A} \cup \mathcal{R}$ and $\mathcal{A} \cap \mathcal{R} = \emptyset$. To detect noncompliance in a given X.509 CCVL implementation, we automatically extract the sets $\mathcal{A}$ and $\mathcal{R}$. Due to the large number of possible certificate chains, explicitly enumerating elements of the sets $\mathcal{A}$ and $\mathcal{R}$ is not feasible. We represent the sets $\mathcal{A}$ and $\mathcal{R}$ symbolically by a set of quantifier-free first order logic (QFFOL) formulas [163] $\{f_1, f_2, \dots, f_n\}$ where each QFFOL formula f_i represents a set of concrete certificate chains. We choose QFFOL as it is sufficiently expressive and also decidable for certain theories (*e.g.*, bitvector, array)—one can leverage an SMT solver to detect noncompliance—whereas the full first order logic (FOL) is undecidable. We use the theory of bitvectors and array.

For a given X.509 implementation, we extract the sets $\mathcal{A}$ and $\mathcal{R}$ by symbolically executing the CCVL of that given implementation with respect to a symbolic certificate chain. Symbolically executing the CCVL can capture the validation logic for that given implementation through path constraints and their associated return values of the CCVL function. The path constraint in question here contains input variables coming from the input certificate chain that has fields and extensions we marked to have symbolic values. Given a collected path constraint f and its associated boolean value b returned by the CCVL function, if $b = \mathsf{true}$ (resp., false), it signifies that any concrete certificate chain c that satisfies the constraint f (*i.e.*, $c \models f$) is accepted (resp., rejected) by the given CCVL. Precisely, after symbolic execution of the CCVL, we have $\mathcal{C} = \{\langle f_1, b_1\rangle, \langle f_2, b_2\rangle, \ldots, \langle f_n, b_n\rangle\}$ where f_i is a path constraint (*i.e.*, QFFOL formula) we obtained during symbolic execution of the CCVL and $b_i \in \{\mathsf{true}, \mathsf{false}\}$ is the return value of the CCVL function for the path constraint f_i. From $\mathcal{C}$, we construct $\mathcal{A}$ and $\mathcal{R}$ in the following way: $\mathcal{A} = \{f_i \mid \langle f_i, \mathsf{true}\rangle \in \mathcal{C}\}$ and $\mathcal{R} = \{f_j \mid \langle f_j, \mathsf{false}\rangle \in \mathcal{C}\}$.

The sets $\mathcal{A}$ and $\mathcal{R}$ induced by a given X.509 CCVL implementation are **the core asset of our noncompliance detection approach**. Given the sets of $\mathcal{A}_{\mathsf{test}}$ and $\mathcal{R}_{\mathsf{test}}$ induced by a CCVL implementation under test I_{test} and the sets $\mathcal{A}_{\mathsf{standard}}$ and $\mathcal{R}_{\mathsf{standard}}$ induced by the X.509 standard specification (*e.g.*, RFC), I_{test} is noncompliant if one of the following (or, both) hold: (1) $\mathcal{A}_{\mathsf{test}} \neq \mathcal{A}_{\mathsf{standard}}$ and (2) $\mathcal{R}_{\mathsf{test}} \neq \mathcal{R}_{\mathsf{standard}}$. For a given I_{test}, we can use its $\mathcal{A}_{\mathsf{test}}$ and $\mathcal{R}_{\mathsf{test}}$ to expose noncompliance in several ways, possibly by leveraging an SMT solver [164, 166]. We discuss them presently.

4.3.3 Approaches for Exposing Noncompliance

We now discuss three approaches where we leverage symbolic execution and the sets $\mathcal{A}$ and $\mathcal{R}$ to find noncompliance in X.509 CCVL implementations.

Noncompliance during Symbolic Execution

During symbolic execution of the X.509 CCVL function of a given implementation, the symbolic execution engine can discover certain low level memory errors (*e.g.*, array out of bounds). We have discovered an erroneous extension processing bug using this approach. We present the details in Section 4.5.

Simple Searching of the Path Constraints

By inspecting all the path constraints in the set $\mathcal{A} \cup \mathcal{R}$ for a particular CCVL implementation, one can easily notice missing checks of certain certificate fields. Let us assume that we assigned the subject name field of a certificate to have the symbolic value sym_sub_name. We can then perform a search with the string sym_sub_name (*i.e.*, often a simple `grep` will suffice) among all the path constraints in $\mathcal{A} \cup \mathcal{R}$. If the search turns up empty, one can conclude with high confidence that the implementation does not check the subject name field. This approach enables exposure of noncompliance due to an implementation's inability to take certificate fields into consideration during the CCVL decision making process. We have discovered several serious noncompliances using `grep`.

Cross Validation

To expose deeper noncompliant instances—the ones due to an implementation's inability to impose proper validity checks on a certificate field even after recognizing it—ideally we want the sets $\mathcal{A}_{\mathsf{standard}}$ and $\mathcal{R}_{\mathsf{standard}}$ induced by the X.509 standard specification. We have, however, neither a formally verified CCVL implementation we can extract the sets $\mathcal{A}_{\mathsf{standard}}$ and $\mathcal{R}_{\mathsf{standard}}$ from, nor a formal specification for X.509 CCVL at our disposal. We compensate for the lack of the sets $\mathcal{A}_{\mathsf{standard}}$ and $\mathcal{R}_{\mathsf{standard}}$ by utilizing the existence of a large number of open source SSL/TLS library implementations. We can perform a cross validation (or, differential testing [58,162]) by pitting the different implementations against each other. If two implementations come to different conclusions about whether a given certificate chain is valid, even though it is not clear which implementation is noncompliant, we can conclude that one of the libraries is noncompliant. Precisely, for any two implementations I_1 and I_2 and their corresponding sets $\mathcal{A}_1$, $\mathcal{R}_1$, $\mathcal{A}_2$, and $\mathcal{R}_2$, any $c \in \mathcal{C}$ such that (1) $c \in \mathcal{A}_1 \wedge c \in \mathcal{R}_2$ or (2) $c \in \mathcal{A}_2 \wedge c \in \mathcal{R}_1$ represents an instance of noncompliance.

One can utilize the path constraints from two different implementations to find inconsistent conclusions in the following two ways. In our analysis, we follow approach 2.

Let us assume for any two given implementations I_p and I_q, we have the following sets:

$$\mathcal{A}_p = \{a_1^p, a_2^p, \ldots, a_n^p\} \qquad \text{(accepting certificate universe of } I_p)$$

$$\mathcal{R}_p = \{r_1^p, r_2^p, \ldots, r_m^p\} \qquad \text{(rejecting certificate universe of } I_p)$$

$$\mathcal{A}_q = \{a_1^q, a_2^q, \ldots, a_s^q\} \qquad \text{(accepting certificate universe of } I_q)$$

$$\mathcal{R}_q = \{r_1^q, r_2^q, \ldots, r_t^q\} \qquad \text{(rejecting certificate universe of } I_q)$$

Approach 1: To detect inconsistencies between I_p and I_q, one can check to see whether either of the following formulas is satisfiable: $\neg(\bigvee_{(1\leq i\leq n)} a_i^p \leftrightarrow \bigvee_{(1\leq j\leq s)} a_j^q)$ and $\neg(\bigvee_{(1\leq i\leq m)} r_i^p \leftrightarrow \bigvee_{(1\leq j\leq t)} r_j^q)$ ($\leftrightarrow$ stands for logical equivalence). The first (resp., second) formula asserts that the accepting (resp., rejecting) paths of I_p and I_q are not equivalent. Any model of either of the formulas will signify a noncompliant instance. We, however, do not utilize this approach to detect noncompliance for the following three reasons: (1) For each satisfiability query the SMT solver will present one model (*i.e.*, one noncompliant instance) even in the presence of multiple noncompliant instances (**We desire as many noncompliant instances instead of just one at a time**); (2) The resulting formulas are large and it may put heavy burden on the SMT solver; (3) Due to the incompleteness caused by techniques used to relieve path explosion, the extracted sets $\mathcal{A}$ and $\mathcal{R}$ may not be exhaustive (*i.e.*, complete), yielding false positives.

Approach 2: In this approach, we first take each accepting path a_i^p from $\mathcal{A}_p$ and each rejecting path r_j^q from $\mathcal{R}_q$ where $1 \leq i \leq n, 1 \leq j \leq t$, and check to see whether the formula $a_i^p \wedge r_j^q$ is satisfiable by consulting an SMT solver. If the formula is satisfiable, it signifies that there is at least one certificate chain that I_p accepts but

I_q rejects. The model obtained for the formula from the SMT solver, can be used to construct a concrete certificate chain signifying an evidence of inconsistency. We can then repeat the same process by taking each accepting path from I_q and each rejecting path from I_p. Note that, multiple pairs may induce inconsistencies due to the same noncompliant behavior and sometimes best-effort manual analysis of the source code is needed to detect the root cause.

4.3.4 Scalability Challenges of Applying Symbolic Execution

The application of symbolic execution in a straightforward way to extract the sets $\mathcal{A}$ and $\mathcal{R}$, considering all certificates in the chains and other arguments to the CCVL function to have symbolic values, will not yield a scalable noncompliance detection approach. *Our feasibility evaluation have verified this observation.* We have also tried only one of the certificates in the chain to have symbolic values and even then the symbolic execution did not finish due to resource exhaustion. The scalability problem is predominantly due to *symbolic value dependent loops*—loops whose terminating conditions depend on symbolic values—in the certificate parsing implementation. One way to get around this challenge is to assume the correctness of the parsing code and just focus on the core CCVL logic. Ignoring the parsing logic, however, is not sufficient to capture the majority of the CCVL logic as some of the sanity checks on the certificate fields are done during parsing. In addition, capturing only the CCVL logic would require one to manually modify the internal data structure where the certificate fields are stored after parsing. This approach requires significant manual efforts (*i.e.*, code comprehension) and is also error-prone.

4.3.5 Our Solution—SymCerts and Problem Decomposition

For addressing the scalability challenge we rely on carefully crafting symbolic certificates and also on our domain specific observations. Rather than extracting the complete sets $\mathcal{A}$ and $\mathcal{R}$, we use domain-specific observations and specially crafted symbolic certificate chains to extract an approximation of the sets $\mathcal{A}$ and $\mathcal{R}$, *i.e.*, $\mathcal{A}_{\mathsf{approx}}$ and $\mathcal{R}_{\mathsf{approx}}$. Our approximation has both under- and over-approximation. To overcome path explosion, we create a chain of **SymCerts** where some portions of each certificate have concrete values whereas the others have symbolic values. SymCerts along with the following observation aid in achieving scalability during the extractions of the sets $\mathcal{A}_{\mathsf{approx}}$ and $\mathcal{R}_{\mathsf{approx}}$ from an X.509 CCVL.

One domain specific observation we use is the **logical independence between certificate fields**. For instance, the logic of checking whether a certificate is expired according to its notAfter field is independent of the logic of checking whether a certificate's issuer name matches with the subject name of the predecessor certificate in the chain. In this case, we can try to capture the logic of checking certificate expiration independently of the checking of issuer and subject names. Based on the notion of independence, we group the certificate fields into equivalence classes where the logic of fields in the same equivalence class should be extracted at the same time, that is, fields in the same equivalence class should be marked to have symbolic values at the same time. We leverage this observation by generating a SymCert chain for each equivalence class where each element of the equivalence class has symbolic values whereas the rest of the fields have concrete values. **Note that we certainly do not claim that the checking logic of all certificate fields are independent**; there are obviously certificate fields whose value influences one another. For instance, the value of the isCA field of an X.509 certificate (*i.e.*, whether the certificate is a CA certificate) prescribes certain corresponding key usage purposes (*i.e.*, affecting the KeyUsage extension). In this case, the isCA field needs to be in the same equivalence class as KeyUsage.

In our analysis, we conservatively partition the certificate fields into 2 equivalence classes. We refer to these two equivalence classes as EqC_1 and EqC_2, respectively. EqC_1 has all the relevant certificate fields symbolic, except the Validity date time period fields which are symbolic only in EqC_2.

4.4 Implementation

In this section, we discuss additional challenges of applying symbolic execution to CCVL code, and our approach to addressing these challenges. We also discuss other aspects of implementing our noncompliance finding approach.

Challenge 1 (Complex Structure of X.509 Certificates)

X.509 certificates are represented in the *Abstract Syntax Notation One* (ASN.1) [167, 168] notation. X.509 certificates are typically transmitted in byte streams encoded following the *DER* (Distinguished Encoding Rules), which are binary in nature. Under the DER format, an X.509 certificate has the form $\langle t, \ell, v \rangle$ where t denotes a type, ℓ denotes the length of the values in bytes, and finally v represents the value. t can represent complex types such as a sequence where the value v can be recursively made of other $\langle t, \ell, v \rangle$ triplets. Such nesting of $\langle t, \ell, v \rangle$ triplets inside a v field can be arbitrarily deep.

The problem of marking the whole certificate byte-stream as symbolic is that, during certificate parsing, the symbolic execution engine will try different values for ℓ as it is symbolic, and the parsing code will keep reading bytes without knowing when to stop. This will cause memory exhaustion.

Approach—SymCerts (Certificates With Symbolic and Concrete Values) To avoid the scalability problem, instead of using a fully symbolic certificate chain, we develop a certificate chain in which each certificate byte-stream contains some of concrete values and some symbolic values. We call each such certificate a **SymCert**.

We construct a SymCert in the following way: For each leaf $\langle t, \ell, v \rangle$ tuple (*i.e.*, v contains a value instead of another $\langle t, \ell, v \rangle$ tuple) in a certificate byte-stream, we ensure that the fields t and ℓ have concrete values whereas only the v field is symbolic. Concrete values of t can be obtained from actual certificates and we use them as the backbone for generating SymCerts. For the l field, we consult the RFC document to select appropriate concrete values. For instance, when marking the OIDs used in the ExtKeyUsage extension symbolic, we give it a concrete length of 8, as most of the standard key usage purposes defined in RFC 5280 [112] are 8-byte long.

Due to the complexity of DER byte-streams, it is difficult for a user to directly manipulate and construct SymCerts from scratch. In addition, due to nesting, changing the length field (*i.e.*, ℓ) of a child $\langle t, \ell, v \rangle$ triplet may require adjustment on the length field (*i.e.*, ℓ) of the parent $\langle t, \ell, v \rangle$ triplet. For this, we developed a Graphical User Interface (**GUI**), by extending the ASN.1 JavaScript decoder [169]. Our GUI allows a user to see and click on different certificate fields, so that they can be replaced with a desired number of symbolic bytes, and the new length will be correctly adjusted. The GUI will then automatically generate code that can be used for symbolic execution. We use OpenSSL to generate concrete certificate chains as the input to our GUI, which constitute the basis of our SymCerts. The philosophy here is that all major fields (*e.g.* optional extensions, criticality booleans) of a certificate need to be explicitly available on the base input certificate, as it is difficult to mark nonexistent fields symbolic.

Challenge 2 (System Time Handling)

Given that our symbolic execution of the implementations would happen at different times, if we simply allow the implementations to use the local system time, then the constraints we have extracted would not be comparable, as the system time elapses.

Approach—Constant Static Time We consider a fixed concrete time value for the system time. We use the same concrete value for these inputs during the analysis of all implementations. Using a symbolic variable is also possible, but using concrete values has the advantage of reducing the complexity of the path constraints which consequently improves scalability.

Challenge 3 (Cryptographic Functions)

The cryptographic functions (*e.g.*, for verifying digital signatures) called by the CCVL contain loops dependent on symbolic data, which severely impact the scalability of symbolic execution.

Approach—Cryptographic Stub Functions We abstract away the cryptographic functions with stub functions. For instance, the function that matches the digital signature of a certificate is abstracted away by a stub function that returns True indicating the match was successful. In this work, we consider cryptographic correctness beyond our scope. Instead, we are interested in finding out what fields are checked and what restrictions are imposed on these fields.

Challenge 4 (Complex String Operations)

As part of the CCVL, implementations are sometimes required to perform complex string operations (*e.g.*, wild card matching, null checking) on certificate fields such as subject name and issuer name. Faithfully capturing the string operations with QF_BVA logic (*i.e.*, QFFOL formulas with equality, bit vector, and array theories)—which is the underlying logic of the symbolic execution engine we use—does not scale well.

Approach—Single Byte Strings We consider names and other string-based certificate fields to have a single byte symbolic value, which significantly improves the scalability. However, because of this, our analysis misses out on finding noncompliance due to erroneous string operations.

Challenge 5 (Hashing for Checking Multi-Field Equality)

When checking the equality of two name fields of certificates—name fields are compound fields containing the following sub-fields such as street address, city, state/province, locality, organizational name & unit, country, common name—some implementations take a hash of the concatenation of all the sub-fields and match the hash values, instead of checking the equality of each sub-field. Trying to solve the constraints from such a match would be similar to attacking the hash collision problem, which is not scalable to analyze with symbolic execution due to symbolic data-dependent loops.

Approach—Hash Stub The hash function in question (*i.e.*, SHA-1) returns a 20-byte hash value. We replace it with a SHA-1 stub which returns a 20-byte value where the (symbolic) name sub-fields are packed together. Because of the single byte approach we introduced to simplify string operations described in the previous challenge, 20-byte is more than enough to pack all name sub-fields of interests.

Challenge 6 (Certificate Chain Length)

While symbolically executing the CCVL of a given implementation, one natural question that arises is: "How many certificates in the symbolic certificate chain should we consider?" An X.509 CCVL implementation often parses the input X.509 certificate chain first and then checks the validity of different fields in the certificates of the parsed chain. During symbolic execution, if the execution detects a loop whose terminating condition relies on a symbolic value, it faces the dilemma of how many times to unroll the loop. Such loops in the implementation often cause path explosion in symbolic execution, resulting in incompleteness and scalability challenges. If we consider the certificate chain length to be symbolic, then the symbolic execution, especially during parsing, would try all possible values for the chain length, causing memory exhaustion.

Approach—Concrete Chain Length For majority of our analysis, we consider a certificate chain of length 3 such that one of the certificates is the root CA certificate, the other is an intermediate CA certificate, and finally the remaining certificate is the certificate of the server currently being authenticated. While analyzing the logic of checking the path length constraint of the basic constraint extension, we also consider certificates with chain length 4 where we have two intermediate CA certificates.

Challenge 7 (Other aspects of Path Explosion)

After the simplifications described above, the symbolic execution engine still generates a large number of paths. We especially observed that making all the v values of $\langle t, \ell, v\rangle$-tuples that represent certificate fields and extensions symbolic yields a lot of paths.

Approach—Early Rejection and Grouping Fields We observed that implementations sometimes do not return early even in the case one of the certificates cannot be parsed or one of the fields validity checks failed. This contributes to a multiplicative factor to the number of paths. We judiciously applied early rejection when parsing or validation check fail. Finally, we applied the *logical independence between certificate fields* based on their semantics to decompose the noncompliance finding fields. We generated two equivalence classes, one consists of time fields related to the certificate Validity period checking, whereas the other contains all the remaining fields. One could possibly employ a more aggressive grouping of fields that need to be check together. We, however, make a conservative choice because if developer incorrectly introduces artificial dependencies in the implementation, we would like to capture them as well.

Challenge 8 (Time Field Comparison)

An X.509 certificate contains two time fields (*i.e.*, notBefore and notAfter) which are compared to the current system time. A time field can be represented in two formats (*i.e.*, GeneralizedTime and UTCTime). In GeneralizedTime, the time field contains a 15-byte ASCII string where day, month, hour, minute, second contribute 2 bytes each; year contributes 4 bytes, and 1 byte is used to represent the time zone.

For UTCTime, the only difference is that year contributes 2 bytes instead of 4. Sanity checks are often performed to ensure the fields are well-formed (*e.g.*, for minute, the most significant digit cannot be larger than 6). Marking the format symbolic and let the symbolic execution engine choose the length of the ASCII string contributes to poor scalability.

Approach—Decomposing Time Fields In addition to checking noncompliance in time fields handling independently from other fields, we further decompose the analysis by analyzing the two time formats separately. We use two different SymCerts during symbolic execution, one with UTCTime and the other with GeneralizedTime, using the concrete length of the date time ASCII string according to the format.

Challenge 9 (Redundant Pair of Paths in Cross-Validation)

When cross-validating two implementations I_p and I_q, the upper bound of discrepancies is $|\mathcal{A}_p| \times |\mathcal{R}_q| + |\mathcal{A}_q| \times |\mathcal{R}_p|$. Based on the number of paths in accepting (*e.g.*, $\mathcal{A}_p$ and $\mathcal{A}_q$) and rejecting (*e.g.*, $\mathcal{R}_p$ and $\mathcal{R}_q$) universes, the maximum number of noncompliance instances can be fairly large which creates a challenge for manual inspection to identify the root cause of the noncompliance.

Approach—Iterative Pruning We observe that many pairwise discrepancies are due to the same root cause. Suppose implementation I_p does not check a particular field that I_q checks. In this case, the missing check in I_p's accepting path will likely be enumerated through many rejecting paths of I_q, resulting in a large number of redundant noncompliance instances. To make it easier to analyze the results of cross-validation, once we have identified such a case, we can concretize the value of that specific field, repeat the extraction step and continue cross-validation with a pruned search space.

Challenge 10 (False Positives)

Due to different domain specific simplifications and the fact that we are abstracting away cryptographic functions, our approach can yield false positives, predominantly due to the path constraint extraction might not be capturing the real execution faithfully. In addition, the specification (*i.e.*, RFC document) states some fields should be checked by a certificate using system, without imposing whether the library or application (the two of them constitute the system) should perform each check. Consequently, SSL/TLS libraries have different API designs due to such unclear separation of responsibility. Some libraries might enforce all the checks during certificate chain validation, while some might not and instead provide optional function calls for application developers desiring such checks, and the other libraries might completely delegate the task of implementing such checks to the application developers. As a clear boundary cannot be drawn easily, false positives can arise if some optional but provided checks are missed out during extraction.

Approach—Concrete Replay To avoid false positives, we use a real client-server setup to help us verify our findings. We capitalize on the fact that a minimalistic sample client code is often made available in the source tree by library developers to demonstrate how the library should be used in application development and use such clients to draw the baseline. To gain confidence that our extracted path constraints adequately capture the real execution, for each accepted (resp., rejected) path constraint, we consult the SMT solver to obtain a concrete certificate chain and feed it to a real client-server setup to see whether the client would actually accept (resp., reject) the chain. This helps us to see whether the real execution concurs with our extraction. Similarly, during cross validation between implementations I_p and I_q, for the discrepancies we found (in the form of a model provided by the SMT solver), we construct a concrete certificate chain out of the model and use the client-server setup to verify it is indeed the case that I_p would accept and I_q would reject the chain.

4.5 Evaluation and Results

We applied our approach in testing 9 open-source implementations from 4 major families of SSL/TLS library source trees, as shown in Table 4.1. Implementations that have been tested in previous study by Brubaker et al. [1] are prefixed with an asterisk. These libraries have seen active deployments in embedded systems and IoT products to satisfy the security needs for connecting to the Internet (*e.g.* axTLS in Arduino [151] and MicroPython [152] for ESP8266, mbedTLS, tropicSSL and MatrixSSL on Particle hardware [149,150], etc.), and are sometimes used even in building applications and libraries on conventional desktop platforms [157–161], due to their performance and small footprint advantage. We test multiple versions of a library from the same family in order to compare with previous work, and to see if the more recent versions implement a more complete and robust validation logic.

In this section we first show statistics that justify the practicality of our approach, and then present noncompliance findings grouped by how we uncovered them along the 3 approaches described in Section 4.3.3, together with other discrepancies and observations that we made while working with the libraries. Findings on recent versions of the implementations, whenever applicable, are reported to the corresponding developers. Many of our reports had led to fixes being implemented in newer versions.

4.5.1 Implementation Efforts and Practicality

For our analysis, we used the KLEE symbolic execution engine [35] and the STP SMT solver [164]. We added around 2000 lines of C++ code for implementing the path constraint extraction and cross validation engines, around 500 lines of Python for parsing path constraints and automating concrete test case generation, and around 400 lines of HTML plus less than 300 lines of JavaScript for the GUI that enables the easy construction of SymCerts.

In order to implement the various optimizations described before, a limited amount of new code need to be added to the libraries that we tested. As shown in Table 4.1,

Table 4.1.: Practicality and efficacy of applying the SymCert approach in testing various small footprint SSL/TLS Libraries

Library (version)	Release Date	Lines of `C` code in library	Lines Added	Paths [EqC_1]	Extraction Time [EqC_1]	Total Paths [EqC_2]	Extraction Time [EqC_2]	Noncompliance Instances Found
axTLS (1.4.3)	Jul 2011	16,283	72	276 (419)	~ 1 minute	≤ 52	≤ 1 minute	7
axTLS (1.5.3)	Apr 2015	16,832	69	276 (419)	~ 1 minute	≤ 52	≤ 1 minute	6
* CyaSSL (2.7.0)	Jun 2013	51,786	33	32 (504)	~ 2 minutes	≤ 26	≤ 1 minute	7
wolfSSL (3.6.6)	Aug 2015	103,690	40	256 (31409)	~ 1 hour	≤ 26	≤ 1 minute	2
tropicSSL (Github)	Mar 2013	13,610	66	16 (67)	~ 1 minute	≤ 30	≤ 1 minute	10
* PolarSSL (1.2.8)	Jun 2013	29,470	66	56 (90)	~ 1 minute	≤ 81	≤ 1 minute	4
mbedTLS (2.1.4)	Jan 2016	53,433	15	13 (536)	~ 1 minute	≤ 41	≤ 1 minute	1
* MatrixSSL (3.4.2)	Feb 2013	18,360	9	8 (160)	~ 1 minute	1	≤ 1 minute	6
MatrixSSL (3.7.2)	Apr 2015	37,879	30	3240 (8786)	~ 1 hour	≤ 25	≤ 1 minute	5

§ The fourth column of the table refers to the lines of code we added to the libraries to make them amenable to our analysis. The fifth and sixth columns display the number of accepting (rejecting) paths we obtained when we made the fields in equivalence class EqC_1 symbolic, and the time it took to complete the extraction process, respectively. The seventh and eighth columns show the upper bound of total paths (including both accepting and rejecting) we observed when the fields in EqC_2 are made symbolic, and the time it took for the path extraction process to complete, respectively.

no more than 75 lines of code were added to each of the library. Most of the new code is used to implement a static system time (see Section 4.4-Challenge 2) and a stub cryptographic signature check (Section 4.4-Challenge 3). Additionally, for CyaSSL 2.7.0, wolfSSL 3.6.6, and MatrixSSL 3.7.2, some code was added to implement the hash stub (see Section 4.4-Challenge 5). PolarSSL 1.2.8 and tropicSSL needed a simplified version of `sscanf()`, and axTLS (both 1.4.3 and 1.5.3) needed a simplified version of `mktime()`, to avoid symbolic-data dependent loops, both of which are used for reading in and converting the format of date-time inputs.

Also shown in Table 4.1 are the performance statistics regarding path extraction. We ran our experiments on a commodity laptop equipped with an Intel i5-2520M CPU and 16GB RAM. Path extraction using EqC_1 for most implementations finished in minutes, while for some heavier ones it completed in hours. The total number of paths ranges from hundreds to the level of ten thousands. For EqC_2, we report the upper bound of the total number of paths, referred to in the table as "Total Paths", because the actual number could vary within each library due to different treatments (and possibly missing checks) for UTCTime and GeneralizedTime (see Section 4.5.3 and 4.5.4 for examples). For each library, extraction using EqC_2 yielded paths at the scale of tens, and finished within a minute.

4.5.2 Errors Discovered By Symbolic Execution

The first opportunity our approach provides is that, during symbolic execution, certain low-level coding issues (*e.g.* memory access errors, division by zeros, etc.) could be found.

Finding 1 (Incorrect extension parsing in CyaSSL 2.7.0[1])

As shown in Listing 4.5.2.i, due to a missing `break` statement after `DecodeAltNames()`, the execution falls through to the next case and also invokes `DecodeAuthKeyId()`. Consequently, some bytes of the subject alternative name extension, which we made symbolic, will overwrite the authority key identifier (a pre-computed hash value) at the time of parsing. The error manifests later during certificate chain validation, when the authority key identifier undergoes some bit shifting operations and modulo arithmetic, effectively turning it into an array accessing index, which is then used to

[1]This bug has been fixed in newer versions of CyaSSL and wolfSSL.

fetch a CA certificate from a table of trusted CA certificates. Since some bytes of the authority key identifier were incorrectly made symbolic during parsing, the execution engine caught potential memory access errors in fetching from the table. This was not reported in [1], which applied fuzzing to test CyaSSL 2.7.0. Our conjecture is that it would be difficult for concrete test cases to hit this bug, as the execution is likely to fall through without triggering any noticeable crashes.

Listing 4.5.2.i: Extension Processing In CyaSSL 2.7.0

```
switch (oid) {
...
case AUTH_INFO_OID:
DecodeAuthInfo(&input[idx], length, cert);
break;
case ALT_NAMES_OID:
DecodeAltNames(&input[idx], length, cert);
case AUTH_KEY_OID:
DecodeAuthKeyId(&input[idx], length, cert);
break;
... }
```

4.5.3 Findings From Simple Search of Path Constraints

Fields of certificates, represented by symbolic variables in our approach, will appear on path constraints if they are involved in branching decisions either directly or indirectly (*e.g.* some other decision variables were calculated based on their values). Consequently, the second opportunity our approach offers is that immediately after extracting path constraints using symbolic execution, missing checks of fields can be discovered by performing “`grep`” on the path constraints.

Finding 2 (pathLenConstraint ignored in CyaSSL 2.7.0, wolfSSL 3.6.6[2])

We noticed that both of the aforementioned libraries fail to take pathLenConstraint into consideration, which means any such restrictions imposed by upper level issuing CAs would be ignored by the libraries.

This was not reported in [1], where fuzzing was applied to CyaSSL 2.7.0. Interestingly, [1] instead reported that CyaSSL 2.7.0 incorrectly rejects leaf CA certificates given the intermediate CA certificate has a pathLenConstraint of 0, and is noncompliant because such certificates should be accepted according to the RFC. Our findings, however, demonstrate that CyaSSL 2.7.0 could not possibly be rejecting certificates for such a reason because it completely ignores pathLenConstraint. Testing CyaSSL 2.7.0 with concrete certificates confirmed our finding. Thus, the conclusion in [1] that CyaSSL 2.7.0 misinterprets RFC regarding pathLenConstraint and leaf CA certificate is incorrect. We conjecture that this is because the frankencerts used as evidence for such conclusion also happen to contain other errors, and were thus rejected by CyaSSL 2.7.0. This demonstrates the difficulty of interpreting results obtained from fuzzing.

Finding 3 (pathLenConstraint of intermediate CA certificates ignored in tropicSSL, PolarSSL 1.2.8[3])

Our path constraints show that even though both tropicSSL and PolarSSL 1.2.8 recognize the pathLenConstraint variable during parsing time, they check only the one that is on the trusted root certificate during chain validation, and ignores those that are on intermediate CA certificates of a given chain.

In addition to the fact that PolarSSL 1.2.8 does not check pathLenConstraint on intermediate CA certificates, another simple search found that PolarSSL 1.2.8 does not check whether the leaf certificate is CA or not (which is not a noncompliant behavior).

[2]wolfSSL 3.9.10 has implemented support for pathLenConstraint [170].

[3]The enforcement of pathLenConstraint from intermediate CA certificates has been introduced since PolarSSL 1.2.18 [171].

It was however reported in [1] that PolarSSL 1.2.8 violates the RFC by always rejecting leaf CA certificates if the intermediate CA certificate has a pathLenConstraint of 0. This is incorrect because PolarSSL 1.2.8 checks neither pathLenConstraint on intermediate CA certificates, nor whether the leaf certificate is CA or not.

Finding 4 (Certain attribute types of distinguished names ignored in axTLS 1.4.3 and 1.5.3)

Both axTLS 1.4.3 and 1.5.3 ignore the country, state/province and locality attribute types of the issuer and subject names. In other words, organizations from different countries and states having the same name would be considered equivalent during matching. This is a clear deviation from RFC 5280 (Section 4.1.2.4) [112].

We have this finding reported to the developer of axTLS, who acknowledged the existence of the problem and implemented a fix in the new 2.1.1 release.

Finding 5 (Inability to process GeneralizedTime in axTLS 1.4.3, tropicSSL)

RFC 5280 (Section 4.1.2.5) [112] states "Conforming applications MUST be able to process validity dates that are encoded in either UTCTime or GeneralizedTime." However, given our SymCerts with GeneralizedTime, both tropicSSL and axTLS 1.4.3 returned only 1 concrete rejecting path with an empty path constraint, hence we conclude that the aforementioned libraries cannot handle GeneralizedTime, which is a non-conformance to the RFC. However, the same SymCerts managed to yield meaningful path constraints in axTLS 1.5.3, showing that support for GeneralizedTime has been added in the newer version of axTLS.

Finding 6 (KeyUsage and ExtKeyUsage being ignored in MatrixSSL 3.4.2, CyaSSL 2.7.0, tropicSSL)

The three aforementioned implementations do not check KeyUsage and ExtKeyUsage extensions. This noncompliance implies that certificates issued specifically for certain intended purposes (*e.g.* only for software code signing) can be used to authenticate a server in SSL/TLS handshakes. Honoring such restrictions imposed by issuing CAs allows the PKI to implement different levels of trust, and help avoid certificate (and CA) misuse in general.

Finding 7 (notBefore ignored in tropicSSL, PolarSSL 1.2.8; validity not checked in MatrixSSL 3.4.2)

Our SymCerts revealed that PolarSSL 1.2.8 does not check the notBefore field, and MatrixSSL 3.4.2 does not have an inbuilt validity check, as there is only 1 path, which is an accepting path with empty constraints, for each of the aforementioned libraries in their respective cases. This is coherent with the findings in [1]. MatrixSSL 3.4.2 delegates the task of checking certificate validity to application developers. tropicSSL has the same problem as PolarSSL 1.2.8, which is not a surprise considering the fact that tropicSSL is a fork of PolarSSL.

Finding 8 (hhmmss of UTCTime ignored in tropicSSL, axTLS 1.4.3 and 1.5.3; hhmmss of both UTCTime and GeneralizedTime ignored in MatrixSSL 3.7.2)

Given UTCTime on certificates, even though axTLS 1.4.3 and 1.5.3 check for both notBefore and notAfter, they do not take the hour, minute and second into consideration, which means that there could be a shift for as long as a day in terms of rejecting future and expired certificates. This finding is particularly interesting for axTLS 1.5.3, as its implementation of GeneralizedTime support can actually handle hour, minute and second, but for some reason UTCTime is processed in a laxer manner. Following our report, the developer of axTLS has acknowledged the problem and is currently considering a fix. Our extracted path constraints show and tropicSSL also suffer from the same problem.

Unlike its older counterpart, MatrixSSL 3.7.2 has implemented validity checks that handle both UTCTime and GeneralizedTime. However, our extracted path constraints revealed that MatrixSSL 3.7.2 does not attempt to check the time portion of the validity fields, regardless of whether the date-time information is in UTCTime or GeneralizedTime. The developers of MatrixSSL had explained to us the decision to ignore the time portion was made due to its embedded origin, where a local timer might not always be available, and in their own words "*having date set correctly is difficult enough*". They have also admitted that as the result of such decision, a 24-hour shift in rejecting future and expired certificates is inevitable.

Finding 9 (notAfter check applies only to leaf certificate in tropicSSL)

Not just that future certificates are not rejected (e.g. missing check for notBefore as described above) in tropicSSL, our path constraints show that, given a chain of certificates, the check on notAfter only applies to the leaf one. This could lead to severe problems, for instance, if a retired private key of an intermediate issuing CA corresponding to an expired certificate got leaked, attackers would be able to issue new certificates and construct a new chain of certificate that will be accepted by tropicSSL.

Finding 10 (Incorrect CA certificate and version number assumptions in axTLS 1.4.3 and 1.5.3, CyaSSL 2.7.0, MatrixSSL 3.4.2)

The aforementioned implementations deviate from the RFC in how they establish whether certificates of various versions are CA certificates or not. As explained previously in Section 4.2.1, in case the certificate has a version older than 3, some out-of-band mechanisms would be necessary to verify whether it is a CA certificate or not. axTLS 1.4.3 and 1.5.3 assume certificates to be CA certificates regardless of the version number. CyaSSL 2.7.0 also does not check the version number, though whenever the basicConstraints extension is present, it will be used to determine whether the certificate is a CA certificate or not. MatrixSSL 3.4.2 does check the version number, and would check the basicConstraints extension for version 3 certificates. However, it would just assume certificates older than version 3 to be CA certificates. The findings on CyaSSL 2.7.0 and MatrixSSL 3.4.2 are coherent with the relevant results reported in [1].

Finding 11 (Unrecognized critical extensions in MatrixSSL 3.4.2, CyaSSL 2.7.0, axTLS 1.4.3 and 1.5.3)

Section 4.2 of RFC 5280 states “A certificate-using system MUST reject the certificate if it encounters a critical extension it does not recognize or a critical extension that contains information that it cannot process.” [112]. Not rejecting unknown critical extensions could lead to interoperability issues. For example, certain entities might define and issue certificates with additional non-standard custom extensions, and rely on the default rejection behavior as described in RFC 5280 to make sure that only a specific group of implementations can handle and process their certificates. However, we found that MatrixSSL v3.4.2 and CyaSSL 2.7.0 would accept certificates with unrecognized critical extensions, which is consistent to the findings in [1].

In addition, we found that axTLS 1.4.3 and 1.5.3 would also accept certificates with unrecognized critical extensions. In fact, based on the path constraints we have extracted, they do not recognize any of the standard extensions that we wanted to test at all, which deviates from RFC 5280, as Section 4.2 says the minimum requirement for applications conforming to the document MUST recognize extensions like key usage, basic constraints, name constraints, and extended key usage, etc. Similarly for mbedTLS 2.1.4, as we have noticed for not implementing support for the name constraints extension, is also noncompliant in that sense. The implication of this is that restrictions imposed by issuing CAs in the form of name constraints will not be honored by mbedTLS 2.1.4, resulting in potential erroneous acceptance of certificates. At the time of writing, developers of mbedTLS have indicated that they currently have no plans on implementing support for this extension, and suggested that application developers can implement their own if desired.

4.5.4 Findings From Cross-Validating Libraries

The final opportunity would be to cross-validate libraries, specifically, for each accepting path of library A and each rejecting path of library B, we perform a conjunction and see if the resulting constraints would be solvable or not. If yes, it signifies a discrepancy exists between the two libraries.

Finding 12 (ExtKeyUsage OID handling in wolfSSL 3.6.6, MatrixSSL 3.7.4)

Our path constraints also unveiled that despite being two of the few libraries that support the extended key usage extension, both wolfSSL 3.6.6 and MatrixSSL 3.7.2 opted for a somewhat lax shortcut in handling the extension: given the object identifier (OID) of a key usage purpose, they do a simple summation (referred colloquially as a non-cryptographic digest function by the developers of MatrixSSL) over all nodes of the OID, and then try to match only that sum. For example, under such scheme, the standard usage purpose "server authentication" (OID 1.3.6.1.5.5.7.3.1, DER-encoded byte values are `0x2B 0x06 0x01 0x05 0x05 0x07 0x03 0x01`) would be treated as decimal 71.

Notice that the extension itself is not restricted to only hold standard usage purposes that are defined in the RFC, and custom key usage purposes are common[4].

[4]For example, Microsoft defines its own key usage purposes and the corresponding OIDs that are deemed meaningful to the Windows ecosystem [172] (the extension is referred to as "Application Policy" in Microsoft terminology, and is not to be confused with "Certificate Policy").

Since OIDs are only meant to be unique in a hierarchical manner, the sums over nodes of OIDs are not necessarily unique. Hypothetically some enterprises under the private enterprise arc (1.3.6.1.4.1) could define OIDs to describe their own key usage purposes, and if added to the extension, those OIDs might be incorrectly treated as some of the standard key usage purposes by the two libraries. This could be problematic for both interoperability and security, as custom key usage purposes would be misinterpreted, and the standard ones could be spoofed.

This finding is a good example of how our approach can be used to discover the exact treatments that variables undergo inside the libraries during execution. It might also be difficult for unguided fuzzing to hit this particular problem.

We contacted the corresponding developers of the 2 libraries regarding this, and both acknowledged the problem exists. wolfSSL has introduced a more rigorous OID bytes checking since version 3.7.3[5], and MatrixSSL is planning to incorporate additional checks of the OID bytes in a new release.

Finding 13 (Incorrect interpretation of `UTCTime` year in MatrixSSL 3.7.2, axTLS 1.4.3 and 1.5.3, tropicSSL)

Since `UTCTime` reserves only two bytes for representing the year, one needs to be cautious when interpreting it. RFC 5280 Section 4.1.2.5.1 [112] says that when the `YY` of a `UTCTime` is larger than or equal to 50 then it should be treated as `19YY`, otherwise it should be treated as `20YY`. This essentially means that the represented range of year is 1950 to 2049 inclusively.

During cross-validation, we noticed that in certain libraries, some legitimate years are being incorrectly rejected (and accepted). A quick inspection of the path constraints, concrete-value counterexamples, and finally the source code, found the following instances of noncompliance.

[5]`https://github.com/wolfSSL/wolfssl/commit/d248a7660cc441b68dc48728b10256e852928ea3`

As shown in Listing 4.5.4.i, MatrixSSL 3.7.2 interprets any `YY` less than 96 to be in the twenty first century. This means certificates that had expired back in 1995 would be considered valid, as the expiration date is incorrectly interpreted to be in 2095. On the other hand, long-living certificates that have a validity period began in 1995 would be treated as not valid yet. The developers acknowledged our report on this and have since implemented a fix in a new release.

Listing 4.5.4.i: UTCTime year adjustment in MatrixSSL 3.7.2

```
y =  2000 + 10 * (c[0] - '0') + (c[1] - '0'); c += 2;
/* Years from '96 through '99 are in the 1900's */
if (y >= 2096) { y -= 100; }
```

Listing 4.5.4.ii: UTCTime year adjustment in tropicSSL

```
to->year += 100 * (to->year < 90);
to->year += 1900;
```

A similar instance of noncompliance was found in tropicSSL, as shown in Listing 4.5.4.ii. tropicSSL interprets any `YY` less than 90 to be in the twenty first century.

Listing 4.5.4.iii: UTCTime year adjustment in axTLS 1.4.3 and 1.5.3

```
if (tm.tm_year <= 50) {    /* 1951-2050 thing */
tm.tm_year += 100;     }
```

A similar issue exists in both axTLS 1.4.3 and 1.5.3. As shown in Listing 4.5.4.iii, there is an off-by-one error in the condition used to decide whether to adjust the year or not. In this case, the year 1950 would be incorrectly considered to mean 2050. Based on the inline comment, it seems to be a case where the developer misinterpreted the RFC. A fix has been implemented in a new version of axTLS following our report.

Finding 14 (Incorrect timezone adjustment in MatrixSSL 3.7.2)

During cross-validation with other libraries, we noticed that the boundary of date checking in the path constraints of MatrixSSL 3.7.2 was shifted by one day. A quick inspection of the date time checking code found that MatrixSSL 3.7.2 uses the `localtime_r()` instead of `gmtime_r()` to convert the current integer epoch time into a time structure. The shift was due to the fact that in conventional `libc` implementations, `localtime_r()` would adjust for the local time zone, which might not necessarily be Zulu, hence deviating from the RFC requirements.

Assuming the date time on certificates are in the Zulu timezone, the implication of this subtle issue is that for systems in GMT-minus time-zones, expired certificates could be considered still valid because of the shift, and certificates that just became valid could be considered not yet valid. Similarly, for systems in GMT-plus time-zones, certificates that are still valid might be considered expired, and future certificates that are not yet valid would be considered valid.

We discussed this with the developers of MatrixSSL. They conjectured the reason for using `localtime_r()` instead of `gmtime_r()` was due to the latter being unavailable on certain embedded platforms. They have agreed, however, as MatrixSSL is gaining popularity on non-embedded platforms, in a new release, they will start using `gmtime_r()` on platforms that support it.

Finding 15 (Overly restrictive notBefore check in CyaSSL 2.7.0[6])

RFC 5280 Section 4.1.2.5 says "The validity period for a certificate is the period of time from notBefore through notAfter, inclusive." However, when cross-validating CyaSSL 2.7.0 with other libraries, from the concrete counterexamples we noticed that discrepancy exists in how the same notBefore values would be accepted by other libraries but rejected by CyaSSL 2.7.0, while such discrepancy was not observed with

[6]This has been fixed in newer versions of CyaSSL and WolfSSL.

notAfter. An inspection of the notBefore checking code yielded the following instance of noncompliance:

Listing 4.5.4.iv: Erroneous “less than” check in CyaSSL 2.7.0

```
static INLINE int DateLessThan(const struct tm* a,
                               const struct tm* b)
{ return !DateGreaterThan(a,b); }
```

Notice that the negation of $>$ is $\leq$, not $<$, which explains why if the current date time happen to be the same as the one described in notBefore, the certificate would be considered future (not valid yet) and rejected. Hence the notBefore checking in CyaSSL 2.7.0 turns out to be overly restrictive than what the RFC mandates.

This is again a new result, comparing to the previous work [1] that also studied CyaSSL 2.7.0. Our conjecture is that given a large number of possible values, it might be difficult for unguided fuzzing to hit boundary cases, hence such a subtle logical error eluded their analysis.

Finding 16 (KeyUsage and ExtKeyUsage being ignored in PolarSSL 1.2.8)

The fact that PolarSSL 1.2.8 does not check KeyUsage and ExtKeyUsage, evaded our simple search approach but was caught during cross-validation, as the implementation actually parses the two extensions, hence some constraints were added as the result of several basic sanity checks happened during parsing. However, during cross-validation, it became clear that apart from the parsing sanity checks, PolarSSL 1.2.8 does not do any meaningful checks on KeyUsage and ExtKeyUsage.

In fact, this resulted in another instance of noncompliance, as PolarSSL 1.2.8 would not reject certificates with KeyUsage or ExtKeyUsage, even if those two extensions were made critical, and it does not perform any meaningful checks apart from merely parsing them. This is an example where a library is intended to handle an extension but was not able to, because of incomplete implementation.

This is consistent with similar results reported in [1], although the finding that PolarSSL 1.2.8 does not check the KeyUsage extension on intermediate CA certificates was not reported in that paper.

Finding 17 (pathLenConstraint of trusted root misinterpreted in tropicSSL)

During cross validation, it became clear to us that, in tropicSSL: (1) on one hand, some accepting paths would allow the pathLenConstraint variable to be 0; (2) on the other hand, some rejecting paths reject because the pathLenConstraint was deemed to be smaller than an unexpectedly large boundary. In both cases, the pathLenConstraint variable appears to have been misinterpreted by tropicSSL.

We suspect that this might be due to the value 0 in the internal parsed certificate data structure is used to capture the case where the pathLenConstraint variable is absent (*i.e.* no limit is imposed). A quick inspection of the parsing code revealed that our suspicion is indeed correct. In fact, the parsing code is supposed to always add 1 to the variable if it is present on the certificate, but a coding error[7] of missing a dereferencing operator (`*`) in front of an integer pointer means that the increment was applied to the pointer itself but not the value, hence the observed behavior described above.

This subtle bug has a severe implication: it completely defeats the purpose of imposing such restriction on a certificate, as a pathLenConstraint of 0 would be incorrectly treated to mean that the chain length could be unlimited.

Finding 18 (Not critical means not a CA in tropicSSL)

During cross validation, we also noticed that when the intermediate CA certificate's basicConstraints extension is set to non-critical, and the isCA boolean is set to `True`, tropicSSL would consider the intermediate CA certificate not a CA certificate. Additionally, in the path constraints, the symbolic variable representing the critical-

[7]This has been fixed in later versions of PolarSSL and mbedTLS.

ity of basicConstraints and the one that represents the isCA boolean are always in conjunction through a logical AND.

A quick inspection found the following problem in the parsing code that handles the basicConstraints extension:

Listing 4.5.4.v: Incorrect adjustment to the isCA boolean in tropicSSL

```
*ca_istrue = is_critical & is_cacert;
```

This interpretation of the basicConstraints extension deviates from the specification, as RFC 5280 says that clients should process extensions that they can recognize, regardless of whether the extension is critical or not. The criticality of basicConstraints should not affect the semantic meaning of attributes in the extension itself. This is an example of a CCVL being overly restrictive.

4.5.5 Other findings

Here we present other interesting findings that are not explicitly noncompliant behaviors deviating from RFC 5280.

Extra 1 (Ineffective date string sanity check in MatrixSSL 3.7.2)

During cross-validation, we noticed that date time byte values in MatrixSSL 3.7.2 are not bounded for exceedingly large or unexpectedly small values. However, in the constraints, we see combinations of whether each byte is too small or not (though not affecting the acceptance decision), which looked suspiciously like a failed lower boundary check. A quick inspection of the certificate parsing code unveiled the snippet shown in Listing 4.5.5.i that is meant to vet a given date string from a certificate, and reject it with a parser error if the values are outside of an expected range. Unfortunately, due to incorrectly using the `&&` operator instead of `||`, the `if` conditions are never satisfiable. This is also proven by the fact that if we symbolically execute the code snippet in Listing 4.5.5.i, all possible execution paths returns 1. Consequently that code snippet would actually never reject any given strings, hence completely defeating the purpose of having a sanity check.

Listing 4.5.5.i: Date-time string sanity check in MatrixSSL 3.7.2

```
if (utctime != 1) {   /* 4 character year */
  if (*c < '1' && *c > '2') return 0; c++; /* Year */
  if (*c < '0' && *c > '9') return 0; c++;
}
if (*c < '0' && *c > '9') return 0; c++;
if (*c < '0' && *c > '9') return 0; c++;
if (*c < '0' && *c > '1') return 0; c++;   /* Month */
if (*c < '0' && *c > '9') return 0; c++;
if (*c < '0' && *c > '3') return 0; c++;   /* Day */
if (*c < '0' && *c > '9') return 0;
return 1;
```

Following our report, the developers of MatrixSSL have acknowledged this is indeed a faulty implementation. Along with other fixes being implemented to make date-time processing more robust, they have decided that this sanity check will no longer be used in newer versions of MatrixSSL.

Extra 2 (notBefore and notAfter bytes taken "as is" in CyaSSL 2.7.0, WolfSSL 3.6.6, axTLS 1.4.3 and 1.5.3)

For the four aforementioned implementations, we noticed during cross-validation that they do not perform any explicit boundary checks on the value of the date time value bytes of notBefore and notAfter, and just assumed that those bytes are going to be valid ASCII digits (*i.e.* 0–9). It is hence possible to put other ASCII characters in the date time bytes and obtain an exceptionally large (small) values for notAfter (notBefore), though this does not seem to be an imminent threat, nor does it violate the RFC, as the RFC did not stipulate what implementations should do.

Extra 3 (Timezone Handling)

Another discrepancy that we have observed during cross-validating path constraints of different libraries was how they impose/assume the time zone of notBefore and notAfter on certificates. Specifically, we notice that mbedTLS 2.1.4 and wolfSSL v2.3.3 would reject certificates that do not have the timezone ending with a 'Z'.

This is possibly due to the fact that RFC 5280 [112] mandates conforming CAs to express validity in Zulu time (a.k.a GMT or Zero Meridian Time) when issuing certificates, regardless of the type being UTCTime or GeneralizedTime. Other implementations like MatrixSSL 3.7.2, axTLS 1.5.3 and PolarSSL 1.2.8 ignore the timezone character and simply assume the Zulu timezone is always being used.

This is arguably an example of under-specification, as it is not clear whether implementations should try to handle (with proper time zone adjustment) or reject certificates with a non-Zulu timezone, since RFC 5280 [112] did not explicitly mandate an expected behavior.

4.6 Discussions

4.6.1 Takeaway for Application Developers

As a takeaway for application developers that need to use SSL/TLS libraries for processing X.509 certificates, a general rule of thumb is to upgrade to newer versions of the libraries if possible. As demonstrated by our findings, newer versions of implementations, even when originated from the same source tree as their legacy counterparts, are better equipped in terms of features and extension handling, as well as in general having more rigorous checks. Holding on to legacy code could potentially hurt both security and interoperability. Unfortunately, regular software patching, particularly for IoT devices, does not seem to happen widespread enough [173].

We understand that due to the needs to optimize for different application scenarios (*e.g.* small footprint for resource constrained platforms), certain features might not be implemented in their entirety as described in the standard specifications. In order to help application developers to better understand the trade-offs and make a more well-informed decision in choosing which SSL/TLS library to use, we believe that one possibility would be to have a certification program that tests for implementation conformance and interoperability, similar to that of the IPv6 Ready Logo Program [174], and the High Definition Logos [175]. For example, an "X.509 Gold" for libraries that implement most required features correctly, and an "X.509 Ready" for libraries that can only handle the bare minimum but are missing out on certain features.

4.6.2 Limitations

Since our noncompliance detection approach critically relies on symbolic execution which is known to suffer from path explosion, especially in the presence of symbolic data-dependent loops, it is deliberately made to **trade away completeness for practicality** (*i.e.*, our approach is not guaranteed to reveal all possible noncompliances in an implementation and can have false negatives).

Our current scope of analysis does not include the logic for checking certification revocation status and hostname matching. As noted in [1], for both revocation status checking and hostname matching, while some libraries provide relevant facilities, some delegate the task to application developers. In addition, a typical implementation of a hostname matching logic uses complex string operations and analyzing these require a dedicated SMT solver with support for the theory of strings [176]. We leave that for future work.

Moreover, as we use concrete values in SymCerts, symbolic execution sway away from rigorously exercising the parsing logic. Though we have uncovered parsing bugs as reported in Section 4.5, our scrutiny on the parsing code is not meant to be comprehensive. Noticeably, low-level memory errors due to incorrect buffer management in the parsing code, as reported in a recent Vulnerability Note [177], can elude our analysis.

4.6.3 Threat to Validity

In some cases during certificate validation, it is not clear who is required to perform the validity check on a field, *i.e.*, the underlying library or the application using the library. The RFC states that some specific validity check must be performed without clearly identifying the responsible party. This unclear separation of responsibilities have resulted in libraries opting for significantly different API designs. We rely on example usage—often come with the source code in the form of a sample client—to draw a boundary for extracting the approximated certificate accepting (and rejecting) universes. Optional function calls to extra checking logics, if not demonstrated in the sample client programs, will be missed by our analysis. Additionally, if some of the checks performed on certificates are being pushed down to a different phase during SSL/TLS handshake instead of the server certificate validation phase, these checks might be missing from our extraction. We rely on the concrete client-server replay setup to catch them and iteratively include them in the extraction.

Our optimization often rely on the expectation that the value of *some* fields are handled in the implementation in an uniform way. For checking validity of fields that can have variable lengths, we assume the implementation treats each regular length (**not corner cases**) uniformly. In addition, we also assume that the semantic independence of certain certificate fields are maintained in the implementation. For instance, we assume that the certificate validity fields are not dependent on any other fields. Although we have observed that this seems to be the case and the RFC supports it, hypothetically a developer can mistakenly create an artificial dependency.

4.6.4 Future Directions

At the time of writing, existing work on reference SSL/TLS implementations [21,22,178] do not include a formally verified X.509 certificate validation logic. Possible future efforts made along this direction on building a high-confidence implementation of X.509 validation can be used as references to help put verdicts on whether behavioral discrepancies found in other implementations are indeed incorrect and noncompliant.

4.7 Conclusion

In this research, we present a novel approach that leverages symbolic execution to find noncompliance in X.509 implementations. In alignment with the general consensus, we observe that due to the recursive nature of certificate representation, an off-the-shelf symbolic execution engine suffers from path explosion problem. We overcome this inherent challenge in two ways: (1) Focusing on real implementations with a small resource footprint; (2) Leveraging domain-specific insights, abstractions, and compartmentalization. We use SymCerts—certificate chains in which each certificate has a mix of symbolic and concrete values—such that symbolic execution can be made scalable on many X.509 implementations while meaningful analysis can be conducted.

We applied our noncompliance approach to analyze 9 real implementations selected from 4 major families of SSL/TLS source base. Our analysis exposed 48 instances of noncompliance, some of which has severe security implications. We have responsibly shared our new findings with the respective library developers. Most of our reports have generated positive acknowledgments from the developers, and led to the implementation of fixes to the said problems in new releases.

5. SYSTEMATICALLY TESTING SEMANTIC CORRECTNESS OF PKCS#1 v1.5 SIGNATURE VERIFICATION

5.1 Introduction

Developing a deployable cryptographic protocol is by no means an easy feat. The journey from theory to practice is often long and arduous, and a small misstep can have the security guarantees that are backed by years of thorough analysis completely undone. Given well-defined cryptographic constructs originated from mathematical problems that are believed to be hard to solve, proving their functional correctness with respect to the relevant assumptions and security models is hardly the end of the journey. Because of the restrictive assumptions used in designing cryptographic constructs, in reality, additional glue protocols are often needed to generalize such constructs into being able to handle inputs of diverse length and formats. Sometimes glue protocols are also used to wrap around cryptographic constructs for exploiting the duality of certain security guarantees to achieve alternative properties. After careful designs have been devised and standardized, it is also necessary for implementations to faithfully adhere to the specification, in order to ensure the retention of the original designed security and functionality goals in actual deployments. Implementations that deviate from the standard and do not achieve the prescribed level of robustness can lead to a plethora of attacks [7, 8, 55, 179].

The PKCS#1 v1.5 signature scheme, surrounding the RSA algorithm, is one such glue protocol that is widely deployed in practice. Used in popular secure communication protocols like SSL/TLS and SSH, it has also been adapted for other scenarios like signing software. Prior work has demonstrated that lenient implementations of PKCS#1 v1.5 signature verification can be exploited in specific settings (*e.g.*, when small public exponents are being used) to allow the forgery of digital signatures without possession of the private exponent nor factorizing the modulus [7, 8, 11–15]. The identification of such implementation flaws, however, has been mostly based on manual code inspection [7, 8]. The focus of this research[1] is thus to develop a systematic and highly automated approach for analyzing *semantic correctness* of implementations of protocols like PKCS#1 v1.5 signature verification, that is, whether the code adheres to and enforces what the specification prescribes.

Our approach. For identifying semantic weaknesses of protocol implementations, we propose to perform symbolic analysis of the software [147]. Directly applying off-the-shelf symbolic execution tools [35, 148] to test PKCS#1 v1.5 implementations, however, suffers from scalability challenges. This is due to the fact that the inputs to such protocols are often structured with variable length fields (*e.g.*, padding), and can sometimes contain sophisticated ASN.1 objects (*e.g.*, metadata).

One might question the applicability of symbolic analysis on implementations of a cryptographic protocol. The key intuition that we leverage in our approach, is that while the underlying mathematics of cryptographic constructs are typically non-linear in nature, which are often difficult to analyze with constraint solvers, the various variable-sized components used in glue protocols like PKCS#1 v1.5 often exhibit linear relations among themselves and with the input buffer (*e.g.*, sum of component lengths should equal to a certain expected value). Using linear constraints stemming from such relations, we can guide symbolic execution into automatically generating many meaningful concolic test cases, a technique we refer to as *meta-level search*.

To further address scalability challenges faced by symbolic execution, we draw insights from the so-called human-in-the-loop idea [180]. With domain knowledge on the protocol design and input formats, human expertise can partition the input space in a coarse-grained fashion by grouping together parts of the input buffer that should be analyzed simultaneously, making them symbolic while leaving the rest concrete.

[1]A shorter version of this chapter was published at The Network and Distributed System Security Symposium (NDSS) 2019 as a conference paper.

A good partition strategy should constrain and guide symbolic execution to focus on subproblems that are much easier to efficiently and exhaustively search than the original unconstrained input space, and hence achieve good coverage while avoiding intractable path explosions due to loops and recursions.

To facilitate root-cause analysis of an identified deviation, we design and develop a *constraint provenance tracking* (CPT) mechanism that maps the different clauses of each path constraint generated by symbolic execution to their source level origin, which can be used to understand where certain decisions were being made inside the source tree. Our carefully designed CPT mechanism has been demonstrated to incur only modest overhead while maintaining sufficient information for identifying the root-cause of deviations in the source code.

Case Study. The PKCS#1 v1.5 signature scheme is a good candidate for demonstrating the effectiveness of our approach in analyzing semantic correctness, as the protocol itself involves diverse glue components. As we will explain later, to our surprise, even after a decade since the discovery of the original vulnerability [8], several implementations still fail to faithfully and robustly implement the prescribed verification logic, resulting in new variants of the reported attack.

Findings. To show the efficacy of our approach, we first use it to analyze 2 legacy implementations of PKCS#1 v1.5 signature verification that are known to be vulnerable. Our analysis identified not only the known exploitable flaws, but also revealed some additional weaknesses. We then analyze 15 recent open-source implementations with our approach. Our analysis revealed that 6 of these implementations (i.e., strongSwan 5.6.3, Openswan 2.6.50, axTLS 2.1.3, mbedTLS 2.4.2, MatrixSSL 3.9.1, and libtomcrypt 1.16) exhibit various semantic correctness issues in their signature verification logic. Our analysis in an existing theoretical framework shows that 4 of these weak implementations are in fact susceptible to novel variants of Bleichenbacher's low-exponent RSA signature forgery attack [7, 8], due to some new forms of weaknesses unreported before. Exploiting these newly found weaknesses, forging a digital signature does not require the adversary to carry out many brute-force trials as described in previous work [7]. Contrary to common wisdom, in some cases, choosing a larger security parameter (*i.e.*, modulus) actually makes various attacks easier to succeed, and there are still key generation programs that mandate small public exponents [181]. One particular denial of service attack against axTLS 2.1.3 exploiting its signature verification weakness can be launched even if no Certificate Authorities use small public exponents. Among the numerous weaknesses discovered, 6 new CVEs have been assigned to the exploitable ones.

Contributions. This research makes the following contributions:

1. We propose and develop a principled and practical approach based on symbolic execution that enables the identification of exploitable flaws in implementations of PKCS#1 v1.5 signature verification. Specifically, we discuss how to enhance symbolic execution with *meta-level search* in Section 5.2.

2. To aid root-cause analysis when analyzing semantic correctness with symbolic execution, we design and implement a constraint provenance tracker; which is of independent interest. We explain in Section 5.3 how this can help identify root causes of observed implementation deviations with only a modest overhead.

3. We demonstrate our approach with a case study on implementations of PKCS#1 v1.5 signature verification. Our analysis of 2 known buggy (Section 5.4.4) and 15 recent implementations (Section 5.5) of PKCS#1 v1.5 not only led to the discovery of known vulnerabilities but also various new forms of weaknesses. We also provide theoretical analysis and proof-of-concept attacks based on our new findings in Section 5.6.

5.2 Symbolic Execution with Meta-level Search

While symbolic execution is a time-tested means for analyzing programs, the practicality challenges that it faces are also well understood. When dealing with complex structured inputs, one strategy to workaround scalability issues is to draw on domain knowledge to strategically mix concrete values with symbolic variables in the (concolic) test input. When done correctly, this should allow symbolic execution to reach beyond the input parsing code (which makes frequent use of loops and recursions) and explore the post-parsing decision making logic.

As explained in previous work [9], inputs like X.509 certificates that are DER-encoded ASN.1 objects, can be viewed as a tree of {*Tag, Length, Value*} triplets, where the length of *Value* bytes is explicitly given. Hence, if all the *Tag* and *Length* are fixed to concrete values, the positions of where *Value* begins and ends in a test input buffer would also be fixed. One can thus generate a few concrete inputs, and manually mark *Value* bytes of interests as symbolic to obtain meaningful concolic test cases. In fact, just a handful of such manually produced test cases managed to uncover a variety of verification problems [9].

However, cryptographic glue protocols like PKCS#1 v1.5 signatures sometimes involve not only an encoded ASN.1 object, but also input components used for padding purposes, where the length is often implicitly given by an explicit termination indicator. In PKCS#1 v1.5, since padding comes before its ASN.1 structure, the extra room gained due to (incorrectly) short padding can be hidden in any later parts of the input buffer, including many leaf nodes of the encoded ASN.1 object. This means there could be many combinations of lengths of components that constitute the input buffer, all meaningful for testing. Consequently, the concretization strategy used in previous work [9] in this case requires a huge amount of manual effort to enumerate and prepare concolic inputs, and would easily miss out on meaningful corner cases.

To achieve a high degree of automation while preserving a good test coverage, we propose to use symbolic variables not only as test inputs, but also to capture some high-level abstractions of how different portions of the test inputs could be mutated, and let the SMT solver decide whether such mutations are possible during symbolic execution. The key insight is that, the lengths of input components used by protocols like PKCS#1 v1.5 exhibit linear relations with each other. For example, the size of padding and all the other components together should be exactly the size of the modulus, and in benign cases, the length of a parent node in an encoded ASN.1 object is given by the sum of the size of all its child nodes. By programatically describing such constraints, symbolic execution can *automatically* explore combinations of possible component lengths, and generate concolic test cases on the fly by mutating and packing components according to satisfiable constraints.

Given that the input formats of many other protocols also exhibit similar patterns, the *meta-level search* technique should be applicable to them as well. We will explain how to fit this technique specifically for PKCS#1 v1.5 signatures and discuss other engineering details in Section 5.4.

5.3 Constraint Provenance Tracking for Easier Root Cause Analysis

In this section, we present the design, implementation, and empirical evaluation of the constraint provenance tracking (CPT) mechanism. CPT aids one to identify the underlying root-cause of an implementation deviation, identified through the analysis of the relevant path constraints generated by symbolic execution. CPT is of independent interest in the context of semantic correctness checking, as it can be used for many other protocols beyond PKCS#1 v1.5.

5.3.1 Motivation

While the logical formulas extracted by symbolic execution capture the implemented decision-making logic of the test target with respect to its inputs, which enable analysis of semantic correctness and provide a common-ground for differential testing as demonstrated by previous work [9], we argue that after discrepancies have been identified, a root-cause analysis from formula level back to code level is non-trivial to perform, as multiple different code locations of an implementation could have contributed to the various constraints being imposed on a specific symbolic variable. This is further exacerbated by the fact that, modern symbolic execution engines, like KLEE for example, would actively simplify and rewrite path constraints in order to reduce the time spent on constraint solving [35].

Take the following code snippet as a running example. Assuming that each `char` is 1-byte long and `A` is a symbolic variable, a symbolic execution engine like KLEE would discover 3 possible execution paths, with the return value being 0, 1, and 2, respectively.

```
int foo( char A ){
  char b = 10, c = 11;
  if (!memcmp(&A, &c, 1))
    return 0;
  if (memcmp(&A, &b, 1))
    return 1;
  return 2;
}
```

```
(Eq 10 (Read w8 0 A))
```

Example 1: A code snippet with 3 execution paths. The path constraint shown above corresponds to the path that gives a return value of 2.

Although the path that returns 2 *falsifies* the two branching conditions due to the `if` statements (i.e., `A`=11 and `A`≠10), in the end, the simplified constraint only contains the falsification of the second branching condition (i.e., `A` ≠10), as shown in the path constraint. This is because the falsification of the second `if` condition imposes a more specific constraint on the symbolic variable than the first one, and a simplification of the path constraints would discard the inexact clauses in favor of keeping only the more specific and restrictive ones (i.e., $\texttt{A} \neq 11 \wedge \texttt{A} = 10 \leftrightarrow \texttt{A} = 10$).

As illustrated by the example above, although the extracted path constraints faithfully capture the implemented logic, using them to trace where decisions were made inside the code is not necessarily straightforward even on a toy example.

In order to make root-cause analysis easier when it comes to finding bugs with symbolic execution, on top of merely harvesting the final optimized path constraints like previous work did [9], we propose a new feature to be added to the execution engine, dubbed *Constraint Provenance Tracking* (CPT). The main idea is that, during symbolic execution, when a new clause is to be introduced, the engine can associate some source level origin (*e.g.*, file name and line number) with the newly added clause, and export them upon completion of the execution. We envision that when it comes to finding root-causes of implementation flaws, this is better than stepping through an execution using a common debugger with a concrete input generated by the symbolic execution. This is because path constraints offer an abstraction at the level of symbolic variables, not program variables. While one might have to mentally keep track of potentially many different program variables and their algebraic relations when stepping with a debugger (especially when entering some generic functions, *e.g.*, a parser), in symbolic execution those are all resolved into constraints imposed on symbolic variables, and CPT offers insights on where did such impositions happen.

5.3.2 Design of CPT

Performance Considerations

While clause origin can be obtained directly from the debugging information produced by compilers, the constraint optimization needs to be handled delicately. On one hand, such optimizations significantly improve the runtime of symbolic execution [35], on the other, they are often irreversible, hindering root-cause analysis. Striving to balance both performance and usability, in our implementation of CPT, we introduce a separate container for path constraints and their source level origins. The intuition behind introducing the separate container is to let the engine continue performing optimization on the path constraints that drive the symbolic execution, so that runtime performance would not suffer significantly, but then the unoptimized clauses and their origins could be used to assist root-clause analysis. This is essentially trading space for time, and as we show later, the memory overhead is modest. We refer to this as CPT v1.0.

Function Filtering

Another interesting consideration in implementing CPT is what constitutes the origin of a clause. Blindly copying source level information corresponding to the current program counter during symbolic execution is possible, but many times this does not result in a meaningful outcome, because most real software systems are designed and implemented in a modular manner using various libraries.

Consider again the path that returns 2 from the running example (i.e., $\texttt{A} \neq 11 \wedge \texttt{A} = 10$), CPT v1.0 would give the following provenance information, where the origins of the clauses are shown to be from the instrumented C standard library which implements the `memcmp()` function:

```
(Eq false
   (Eq 11 (Read w8 0 A)))  @libc/string/memcmp.c:35
(Eq 10 (Read w8 0 A))      @libc/string/memcmp.c:35
```

While this is technically accurate, from the perspective of analyzing the semantic correctness of a protocol implementation, this is not particularly meaningful. In such a setting, one would most likely not be very interested in analyzing the implementation of the underlying low-level library (*e.g.* the C standard library) and would prefer to have instead the caller of `memcmp()` to be considered as the origin of the clauses.

To this end, we propose to trace stack frames and filter out functions that one would like to ignore in tracking origins of clauses. One can, for example, configure the CPT to not dive into functions from the C standard library through blacklisting exported functions known from the API, and track instead the caller of those functions as the clause origins, which would produce the following CPT output for the same path that returns 2, clearly more useful in understanding the semantics of a protocol implementation:

```
(Eq false
   (Eq 11 (Read w8 0 A)))  @Example1.c:3
(Eq 10 (Read w8 0 A))      @Example1.c:5
```

In addition to the C standard library, we have observed that several cryptography implementations have their own shim layers mimicking the standard library functions (*e.g.* `OPENSSL_memcmp()` in OpenSSL). This is often done for the sake of platform portability (*e.g.* use the C standard library and some platform-specific extensions if they are available, and use a custom imitation if they are not), and is sometimes used to provide custom constant-time alternatives to avoid timing side-channel leakages. All these additional functions can be filtered similarly in CPT as well.

We note that when filtering function calls, there are two possible heuristics. (1) One is to consider the most recent caller of the blacklisted library functions as the clause origin. (2) Another alternative is to consider function calls to have a boundary, where once a blacklisted function has been called, the execution stays in a blacklisted territory until that function returns. While the first heuristic is better at handling callback functions, *we have chosen heuristic 2*, because fully blacklisting all the library functions that CPT should not dive into (or, equivalently, whitelisting all the possible origin functions from a protocol implementation) could be complicated. For example, specific implementations of C standard libraries may use their own undocumented internal functions to implement functions that are exported in the API. Acquiring this knowledge ahead of time could be laborious and hinders generalization.

We use CPT v2.0 to refer to the CPT with function filtering heuristic 2. In the end, we implemented CPT v2.0 by adding less than 750 lines of code to the KLEE toolchain. We chose KLEE as our symbolic execution engine because it is widely used, robust, and is actively maintained.

5.3.3 Performance Evaluation

We now evaluate the performance of KLEE [35] equipped with CPT, and compare it with vanilla KLEE. The goal of this evaluation is to demonstrate that both the memory and runtime overheads induced by the CPT feature are tolerable, as a significant increase in either of the two would severely hinder the practicality of using KLEE in software testing.

The overheads are reported by measuring *time* and *memory* needed by KLEE (with and without CPT) to symbolically execute a suite of target programs. Following what had been previously investigated in the original KLEE paper [35], we use the GNU coreutils package[2] for our evaluation, which consists of various basic tools like `cat` and `ls` used on many Unix-like operating systems. Over the years, coreutils itself has been tested extensively, so we do not intend to find new bugs or achieve a higher code coverage in our experiments.

We follow the experiment setup [182] used in the KLEE paper [35] to run 2 different versions of KLEE on coreutils version 6.11 [183], that is, the original version of KLEE, and the one with CPT v2.0. For each version, we repeat the execution on each coreutil program 3 times and report the average values of runtime and memory measurements. The experiments were conducted on a machine powered by an Intel Core i7-6700 3.40GHz CPU and with 32GB RAM. Table 5.1 shows our measurements on the first 30 programs in coreutils.

To obtain measurement numbers in each experiment, we use the `klee-stat` tool provided by KLEE toolchain. For memory usage, we report both the peak (*maxMem*) and average consumption (*avgMem*), averaged over the 3 executions. Since some of the target programs need an enormous amount of time to finish, following previous work [35, 182], we halt an execution after 1 hour, which explains why some programs in Table 5.1 have a total runtime of about 3600 seconds (*e.g.*, `base64`, `cat`, and `chcon`). In such cases, the mere total execution time is insufficient in showing the time overhead. Hence we also report the average number of completed paths during the 3 executions, which can be used to compare the runtime efficiency of the different versions of KLEE.

To make the number of completed paths comparable, and since we are not focused on code coverage, we also changed the search heuristic used by KLEE into a depth-

[2]`https://www.gnu.org/software/coreutils/coreutils.html`

Table 5.1.: Performance evaluation of KLEE with CPT (Average over 3 trials)

Program	KLEE version	Paths Completed	Time (s)	maxMem (MB)	avgMem (MB)	Program	KLEE version	Paths Completed	Time (s)	maxMem (MB)	avgMem (MB)
[	*Original*	1789	63.06	29.75	27.01	df	*Original*	5016.33	3,600.13	71.27	48.18
	CPT v2.0	1789	62.76	29.79	27.07		*CPT v2.0*	4127.33	3,600.10	68.98	47.64
base64	*Original*	2097957	3,600.01	41.42	34.57	dircolors	*Original*	1019074.33	3,600.02	25	24.64
	CPT v2.0	2091665	3,600.01	41.44	34.65		*CPT v2.0*	1007623.67	3,600.03	25.21	24.77
basename	*Original*	14070	9.2	22.81	22.59	dirname	*Original*	4167	8.42	23.43	22.93
	CPT v2.0	14070	9.15	22.88	22.64		*CPT v2.0*	4167	8.52	23.53	22.98
cat	*Original*	2261170.67	3,600.01	23.68	23.24	du	*Original*	179.67	3,600.55	55.21	43.97
	CPT v2.0	2248991.67	3,600.01	23.76	23.32		*CPT v2.0*	179.33	3600.52	55.32	43.65
chcon	*Original*	480351	3,600.02	59.75	57.82	echo	*Original*	5134030	3,600.01	22.6	22.42
	CPT v2.0	477896	3,600.01	59.95	57.94		*CPT v2.0*	5081012	3,600.01	22.91	22.54
chgrp	*Original*	705117.33	3,600.04	479.42	286.97	env	*Original*	508649	942.72	24.04	23.07
	CPT v2.0	703403.33	3,600.05	478.46	288.03		*CPT v2.0*	508649	925.12	24.13	23.15
chmod	*Original*	430347	3,600.05	393.3	221.09	expand	*Original*	3466952	3,600.01	63.96	54.49
	CPT v2.0	427392.33	3,600.12	378.08	211.71		*CPT v2.0*	3377675.67	3,600.01	64.05	54.36
chown	*Original*	550473.67	3,600.06	353.46	201.4	expr	*Original*	4653	3,600.09	363.95	212.91
	CPT v2.0	543620.67	3,600.04	349.93	200.03		*CPT v2.0*	4645	3,600.14	363.82	212.58
chroot	*Original*	1496	7.25	23.83	23.16	factor	*Original*	618634.33	3,600.06	381.06	208.15
	CPT v2.0	1496	7.58	23.98	23.23		*CPT v2.0*	619403.33	3,600.06	381.57	208.66
cksum	*Original*	2552	7.91	24.81	23.64	false	*Original*	23	0.08	20.85	20.68
	CPT v2.0	2552	7.74	24.85	23.75		*CPT v2.0*	23	0.08	20.9	20.72
comm	*Original*	3895174.33	3,600.01	92.59	68.91	fmt	*Original*	1330	3,610.77	25.88	25.36
	CPT v2.0	3857897.33	3,600.01	91.86	68.51		*CPT v2.0*	1308	3,610.72	25.96	25.42
cp	*Original*	497	3,625.05	28.76	28.37	fold	*Original*	4498176	3,600.01	23.46	23.2
	CPT v2.0	496.33	3,615.35	28.81	28.44		*CPT v2.0*	4426844	3,600.01	23.56	23.28
cut	*Original*	3824504.33	3,600.01	26.21	25.76	head	*Original*	1445240	3,600.02	2,073.72	1,496.51
	CPT v2.0	3826345	3,600.01	26.31	25.84		*CPT v2.0*	1445458	3,600.03	2,073.82	1,497.35
date	*Original*	3564.67	3,602.71	60.79	39.59	hostid	*Original*	1022352	3,600.01	25.79	25.46
	CPT v2.0	3560	3,602.82	61.21	39.6		*CPT v2.0*	1021386.67	3,600.04	25.99	25.59
dd	*Original*	1069290.67	3,600.02	26.48	26.07	hostname	*Original*	991770.67	3,600.01	25.23	24.89
	CPT v2.0	1075813.33	3,600.02	26.67	26.19		*CPT v2.0*	994186.67	3,600.01	25.4	25.01

first search (DFS), instead of a random search as prescribed by the recipe [182], to avoid non-determinism. We also increased the maximum memory usage for each execution to 16GB from the prescribed 1GB [182]. However, as can be seen in Table 5.1, none of the tested programs approached close to this limit.

All in all, the two versions of KLEE yielded comparable total runtime (or, paths completed) and memory usages. CPT v2.0 in general consumes a little more memory and is slightly slower than the original KLEE, though the overheads are insignificant. In the rest of this chapter, unless explicitly mentioned, we are using KLEE with CPT v2.0 by default.

5.4 A Case Study on PKCS#1 v1.5 RSA Signature Verification

We center our analysis around the problem of PKCS#1 v1.5 signature verification. This is particularly suitable for showcasing the merit of enhancing symbolic execution with meta-level searching, as it features diverse glue components including explicitly terminated padding with implicit length, as well as a sophisticated ASN.1 structure. Despite the PKCS#1 family has newer algorithms like RSA-PSS [RFC8017], the v1.5 signature scheme continues to be widely-used in Web PKI and other security-critical network protocols like *SSH* [RFC4253] and *IKEv2* [RFC7296] for authentication purposes.

5.4.1 Technical Background

In this section, we provide a brief overview of RSA signature verification while using PKCS#1 v1.5 as the padding scheme. For the ease of exposition, we provide a list of the notations we use and their meaning in Table 5.2.

Following the usual RSA notations, we use d, e, and n to denote the RSA private exponent, public exponent, and modulus, respectively. $\langle n, e \rangle$ constitutes an RSA public key. We use $|n|$ to denote the size of the modulus in bits. Suppose m is the message for which an RSA signature is to be generated. In the context of X.509 certificates (and CRLs), m would be the ASN.1 DER-encoded byte sequence of `tbsCertificate` (and `tbsCertList`) [RFC5280].

Benign signature generation. For generating an RSA signature of message m in accordance to PKCS#1 v1.5, the signer first computes the hash of m, denoted $H(m)$, based on the hash algorithm of choice (e.g., SHA-1). Then, $H(m)$ and the corresponding meta-data identifying the used hash algorithm and other relevant parameters (if any) are packed into an ASN.1 DER-encoded structure. The necessary amount of padding and other meta-data are prepended to the ASN.1 structure to create a structured input I of size $|n|$, which is then used as an input to the signer's

Table 5.2.: Notation used in our discussion on PKCS#1 v1.5

Symbol	Description	Symbol	Description
n	RSA modulus	e	RSA Public Exponent
d	RSA Private Exponent	$\|n\|$	length of modulus in bits
m	message to be signed	m_v	message received by verifier
I	formatted input to the signer's RSA operation		
S	Signature, $S \equiv I^d \bmod n$ in benign cases		
O	verifier's RSA output, $O \equiv S^e \bmod n$		
$H(m_s)$	signer's version of $H(m)$, contained inside O		
$H(m_v)$	verifier's computed hash of m_v		
I_v	verifier's construction of I given $H(m_v)$		
Symbol	**Description**	**Symbol**	**Description**
`BT`	Block Type	`PB`	Padding Bytes
`AS`	ASN.1 Structure, containing $H(m_s)$		
w	ASN.1 Length of `AS.DigestInfo`		
u	ASN.1 Length of algorithm OID		
x	ASN.1 Length of `AlgorithmIdentifier`		
y	ASN.1 Length of parameters		
z	ASN.1 Length of `Digest`		

RSA operation. The exact format of I is discussed below. Then, the signature will be $S = I^d \bmod n$.

Signature verification. Upon receiving a signed object (say an X.509 certificate), the verifier parses S from it and computes $O := S^e \bmod n$, where O represents the output of the verifier's RSA operation, formatted just like I in correct cases. Given m_v (say `tbsCertificate` of a received certificate), the verifier then computes $H(m_v)$ and compare it against the $H(m_s)$ contained in O. Like previous work has discussed [7], this comparison could be done in the following two manners.

Construction-based verification. Using this approach, the verifier takes $H(m_v)$ and prepares I_v, similar to how the signer is expected to prepare I prior to signing. If $I_v \equiv O$ then the signature is accepted.

Parsing-based verification. Many implementations seem to prefer a parsing-based approach, and this is where things can potentially go wrong. In essence, the goal of this approach is to parse $H(m_s)$ out of O. Many parsers are, however, too lenient even when O is malformed, which gives room for the so-called Bleichenbacher low-exponent brute-force attack.

Structured input (I) and output (O) format. In the benign case, I and O should be formatted as follows:

`0x00 || BT || PB || 0x00 || AS`

`BT` is often referred to as the block type [RFC2313], and `PB` represents the padding bytes. For the purpose of signature generation and verification, `BT` $\equiv$ `0x01` and `PB` $\equiv$ `0xFF 0xFF` ... `0xFF`. Additionally, `PB` has to be at least 8-byte long, and also long enough such that there would be no extra bytes following `AS`. The `0x00` after `PB` signifies the end of padding. `AS` is an ASN.1 DER-encoded byte stream that looks like this (assuming $H()$ being SHA-1):

```
/** all numbers below are hexadecimals **/
/* [AS.DigestInfo] */
30 w                        // ASN.1 SEQUENCE, length = w
    /* [AlgorithmIdentifier] */
    30 x                    // ASN.1 SEQUENCE, length = x
        06 u 2B 0E 03 02 1A     // ASN.1 OID, length = u
        05 y          // ASN.1 NULL parameter, length = y
    /* [Digest] */
    04 z                 // ASN.1 OCTET STRING, length = z
      /* H(m), H()=SHA-1(), m = "hello world" */
      2A AE 6C 35 C9 4F CF B4 15 DB
      E9 5F 40 8B 9C E9 1E E8 46 ED
```

Since DER encoded ASN.1 structures are essentially a tree of {*Tag, Length* ,*Value*} triplets, the length of a parent triplet is defined by the summation of the length of its child triplets. Assuming SHA-1, we can derive the following semantic relations among the different length variables for benign cases: $u = 5$; $z = 20$; $x = 2+u+2+y$; $w = 2 + x + 2 + z$.

For most common hash algorithms like MD5, SHA-1, and the SHA-2 family, the algorithm parameter has to be `NULL` and $y \equiv 0$ [RFC2437, RFC4055]. Historically there were confusions on whether the `NULL` algorithm parameter can be omitted, but now both explicit `NULL` and absent parameters are considered to be legal and equivalent [RFC4055]. This could be a reason why some prefer parsing-based over construction-based, as in the latter approach the verifier would have to try at least two different constructions $\{I_{v_1},\ I_{v_2}\}$ to avoid falsely rejecting valid signatures. We focus on the explicit `NULL` parameter case in this research, as it had been shown that the lenient processing of the parameter bytes can lead to signature forgery [7], and rejecting absent parameter is a compatibility issue easily identifiable with one concrete test case.

When PKCS#1 v1.5 signatures are used in other protocols like SSH and IKEv2 not involving X.509 certificates, the aforementioned steps work similarly with a different input message m (*e.g.*, m could be the transcript containing parameters that were exchanged during a key exchange algorithm).

5.4.2 Testing Deployed Implementations with Our Approach

We now discuss the different challenges and engineering details of how to make the implementations amenable to symbolic analysis. As discussed before, we use KLEE with CPT as our choice of symbolic analysis tool. Building an implementation for KLEE generally takes a few hours of trial-and-error to tune its build system into properly using LLVM.

Scalability Challenges

Since the length of O is given by $|n|$, for the best coverage and completeness, ideally one would test the verification code with a $\frac{|n|}{8}$-byte long symbolic buffer mimicking

O. For implementations that use the parsing-based verification approach, however, since there are possibly many parsing loops and decisions depend on values of the input buffer, using one big symbolic buffer is not scalable.

To workaround scalability challenges, we use a two-stage solution. We first draw on domain knowledge to decompose the original problem into several smaller subproblems, each of which symbolic analysis can then efficiently and exhaustively search. Then for each subproblem we apply the meta-level search technique to automatically generate concolic test cases.

Stage 1. Coarse-grained decomposition of input space. In the first stage, we partition the input space influencing the exploration of the PKCS#1 v1.5 implementations in a coarse-grained fashion. Our coarse-grained partitioning resulted in three partitions, each corresponds to a dedicated test harness. For each implementation, the 3 test harnesses focus on testing various aspects of signature verification while avoiding scalability challenges. Across different implementations, each of the 3 test harnesses—denoted {*TH1, TH2, TH3*}—is focused on the same high-level aspect of testing. The test harnesses would invoke the implementations' PKCS#1 v1.5 signature verification functions, just like a normal application does. Depending on the API design of a specific implementation, the test harnesses also provide the appropriate verification parameters like an RSA public key, $H(m_v)$ (or in some cases, m_v directly) and a placeholder signature value.

Among the different harnesses, *TH1* is designed to investigate the checking of `BT`, `PB`, z the length of $H(m_s)$, and the algorithm parameters, while *TH2* is geared towards the matching of OID in `AlgorithmIdentifier`. Both *TH1* and *TH2* use a varying length of `PB` but the ASN.1 length variables u, w, x, y, z are kept concrete. In contrast, *TH3* has everything else concrete, reminiscent of a correct well-formed O, but u, w, x, y, z are made symbolic, to see how different length variables are being handled and whether an implementation would be tricked by absurd length values. In general, loops depending on unbounded symbolic variables poses threats to termination, however, as we would discuss below, in the context of PKCS#1 v1.5 signatures, one can assume all the length variables are bounded by some linear functions of $|n|$ and still achieve meaningful testing.

Stage 2. Meta-level search using relations between glue components. Following the meta-level search idea discussed in Section 5.2, in both *TH1* and *TH2*, we provide linear constraints that describe the relations between $w, x, y, z, \frac{|n|}{8}$ and $|\texttt{PB}|$. As such, during symbolic execution, many different possible concolic test input buffers would be packed with respect to the given constraints in *TH1* and *TH2*, which effectively expand the two test harnesses automatically into many meaningful test cases, without the need to manually craft a large number of test harnesses, one for each test case. This is essentially a form of *model counting*. Including effort of studying the PKCS#1 v1.5 specification, developing the meta-level search code for {*TH1, TH2*} took a few days. This is however a one-time effort, as the code is generic and was reused across all implementations that we tested. Finally, *TH3* covers the extra cases where w, x, y, z are not constrained in terms of each other and $\frac{|n|}{8}$.

Memory Operations with Symbolic Sizes

We note, however, performing memory allocation and copy (*e.g.*, `malloc()` and `memcpy()`) with symbolic lengths would result in a concretization error where KLEE would try to concretize the length and continue the execution with one feasible concrete length value, hence missing out on some possible execution paths.

Explicit loop introduction. To avoid such concretization errors, when implementing the meta-level search in *TH1* and *TH2*, we use some simple counting `for`-loops, as shown below, to guide KLEE into trying different possible values of the symbolic lengths. What happens is that for each feasible value (with respect to known constraints that are imposed on those symbolic variables), KLEE would assign it to k and fork the execution before the memory allocation and copy, hence being able to try different lengths and not cutting through the search space due to concretization.

```
size_t k; for (k = 0; k < sym_var; k++){}
```

```
/** execution forks with possible values of k **/
dest = malloc(k);      // k already concretized
memcpy(src, dest, k); // k already concretized
```

Bounding parameter length. Since explicit loop introduction is essentially trading time and space for coverage, it will not work practically if the range of possible values is very large. Fortunately, in PKCS#1 v1.5, the size of O is bounded by $|n|$. We leverage this observation to make our symbolic analysis practical, by focusing on a small $|n|$. Specifically, in our test harnesses, we assume the SHA-1 hash algorithm, as it is widely available in implementations, unlike some other older/newer hash functions, and that $|n|$ is 48-byte long (except for MatrixSSL, explained later), so that even after the minimum of 8-byte of `PB` there would still be at least 2 bytes that can be moved around during testing. Though in practice a 384-bit modulus is rarely used, and SHA-1 is now considered weak and under deprecation, since $|n|$ and the hash algorithm of choice are just parameters to the PKCS#1 v1.5 signature verification discussed in Section 5.4.1, assuming uniform implementations, our findings should be extensible to signatures made of a larger $|n|$ and other hash algorithms.

Accessing relevant functions for analysis

Finally, in order to make the implementation amenable to symbolic execution, one would need a customary, minuscule amount of modifications to the source tree. In this case, the modifications are made mainly to (1) change the visibility of certain internal functions; (2) inject the test buffer into the implementation's verification code. Test buffer injection is typically added to the underlying functions that implement the RSA public key operation which compute $O := S^e \bmod n$, easily identifiable with an initial test harness executed in an instrumented manner. Writing the test harnesses and adding the modifications generally take a few hours. In the case of unit tests (and stub functions) for signature verification are readily available (*e.g.* in Openswan), we can simply adapt and reuse their code.

5.4.3 Identifying semantic deviations

Path constraints extracted by symbolic execution can be analyzed in the following two ways to identify implementation flaws. When testing recent implementations, we would use both. Recall that PKCS#1 v1.5 is a deterministic padding scheme and we focus on the explicit `NULL` parameter case. For each test harness, if more than one accepting paths can be found by symbolic execution, then the implementation is highly likely to be deviant. (1) With CPT, one can inspect the path constraints and the origins of their clauses, as well as the generated test cases, to identify the faulty code. (2) To help highlight subtle weaknesses, we adopt the principle of differential testing [58] by *cross-validating* path constraints of different implementations, similar to previous work [9].

5.4.4 Feasibility Study

To validate the efficacy of our approach, we first apply it to test historic versions of OpenSSL and GnuTLS that are known to exhibit weaknesses in their signature verification, without using differential cross-validation for fairness reasons. The summary of results can be found in Table 5.3.

As expected, both OpenSSL 0.9.7h and GnuTLS 1.4.2 use the parsing-based approach for verification. In fact, because both of them also perform some memory allocations based on parsed length variables that are made symbolic in *TH3*, so they both needed explicit loop introduction as discussed before.

For OpenSSL 0.9.7h, the numerous accepting paths in *TH1, TH2* can be attributed to the fact that it accepts signatures containing trailing bytes after `AS`, which is exactly the original vulnerability that enables a signature forgery when $e = 3$ [8, 184]. On top of that, with *TH3*, we found that in addition to the one correct accepting path, there exists other erroneous ones. Specifically, we found that for the ASN.1 length variables y and z, besides the benign values of $y = 0$ and $z = 20$, it would also accept $y = 128$ and $z = 128$, which explains why there are four accept-

Table 5.3.: Result Summary of Testing Known Vulnerable PKCS#1 v1.5 Implementations with Symbolic Execution

Implementation (version)	**Test Harness**	**Lines Changed**	**Execution Time ‡**	**Total Paths (Accepting)**
GnuTLS (1.4.2)	*TH1*	6	00:01:32	2073 (3)
	TH2		01:03:12	127608 (21)
	TH3	8	00:07:35	1582 (1)
OpenSSL (0.9.7h)	*TH1*	4	00:07:23	4008 (3)
	TH2		00:00:46	1432 (3)
	TH3	6	00:33:24	3005 (4)

‡ Execution Time measured on a commodity laptop with an Intel i7-3740QM CPU and 32GB DDR3 RAM running Ubuntu 16.04.

ing paths. This is due to the leniency of the ASN.1 parser in OpenSSL 0.9.7h, which when given certain absurd length values, it would in some cases just use the actual number of remaining bytes as the length, yielding overly permissive acceptances during verification. Though not directly exploitable, this is nonetheless an interesting finding highlighting the power of symbolic analysis, and we are not aware of prior reports regarding this weakness.

For GnuTLS 1.4.2, the multiple accepting paths induced by *TH1* are due to the possibility of gaining extra free bytes with an incorrectly short padding and hiding them inside the algorithm parameter part of `AS`, which will then be ignored and not checked. This is the known flaw that enabled a low-exponent signature forgery [7,14]. Additionally, with *TH3*, we found that there exist an opportunity to induce the parser into reading from illegal addresses, by giving u a special value. Specifically, assuming SHA-1, after the parser has reached but not consumed u, there are still 30 bytes remaining in `AS`. Despite the several sanity checks in place to make sure that the parsed length cannot be larger than what is remaining, by making u exactly 30, it does not violate the sanity checks, but at a later point when the parser attempts to read the actual OID value bytes, it would still be tricked into reading beyond `AS`, which resulted in a memory error caught by KLEE.

The 21 accepting paths (1 correct and 20 erroneous) induced by *TH2* in GnuTLS 1.4.2 can be attributed to how the parser leniently handles and accepts malformed algorithm OIDs. According to the X.690 standard [185], to encode an OID, from the third node onward, each node would take one byte if it is not greater than 127 (short form). If a node is larger than 127 (long form), then for all its encoded bytes except the last byte, the most significant bit would be 1, and the actual value would be the concatenation of the least significant 7 bits of all the encoded bytes (including the last byte where the high bit is off). In such an encoding scheme, one can turn a short form (*e.g.*, `0x0E`) into a long form by prepending meaningless bytes of 128 in front (*e.g.* `0x80 0x80 0x0E`), as the decoding results would be exactly the same, though the standard specification is explicitly against it [185]. Since for the OID of SHA-1 (1.3.14.3.2.26), all nodes from the third one onward are less than 128, they would all be encoded using the short form. However, our testing has discovered automatically that due to the leniency of the ASN.1 parser used by GnuTLS 1.4.2, it is willing to accept the meaningless long form. As the result of which, one can use an incorrectly short padding and spend the extra bytes gained in prepending `0x80` in front of encoded bytes (except the first one). Since our *TH2* allows for at most 2 bytes due to short padding to be moved around, and there are 5 locations in the encoded OID of SHA-1 where the meaningless `0x80` can be inserted (from before `0x0E` to after `0x1A`), the total number of accepting paths is $\binom{5}{0} + \binom{5}{1} + \binom{6}{2} = 21$, with 1 correct and 20 erroneous. This over-permissiveness in signature verification does not seem to have been reported before.

By both recreating known vulnerabilities and finding new weaknesses in the old versions of GnuTLS and OpenSSL, we have demonstrated the efficacy of our proposed approach.

Table 5.4.: Result Summary of Testing various New PKCS#1 v1.5 Implementations with Symbolic Execution

Implementation (version)	Test Harness	Lines Changed	Execution Time †	Total Paths (Accepting)‡	Implementation (version)	Test Harness	Lines Changed	Execution Time †	Total Paths (Accepting)‡
axTLS (2.1.3)	*TH1*	7	01:42:14	1476 (6)	MatrixSSL (3.9.1) *CRL*	*TH1*	4	00:01:55	4574 (21)
	TH2		00:00:05	21 (21)		*TH2*		00:00:04	202 (61)
	TH3	9	00:00:10	21 (1)		*TH3*		00:00:07	350 (7)
BearSSL (0.4)	*TH1*	3	00:01:55	3563 (1)	mbedTLS (2.4.2)	*TH1*	7	00:14:56	51276 (1)
	TH2		00:00:06	42 (1)		*TH2*		00:00:03	26 (1)
	TH3		00:00:00	6 (1)		*TH3*		00:00:00	38 (1)
BoringSSL (3112)	*TH1*	3	00:06:09	3957 (1)	OpenSSH (7.7)	*TH1*	6	00:07:00	3768 (1)
	TH2		00:00:08	26 (1)		*TH2*		00:00:08	22 (1)
	TH3		00:00:00	6 (1)		*TH3*		00:00:00	2 (1)
Dropbear SSH (2017.75)	*TH1*	4	00:46:10	1260 (1)	OpenSSL (1.0.2l)	*TH1*	4	00:06:31	4008 (1)
	TH2		00:00:11	23 (1)		*TH2*		00:00:56	1148 (1)
	TH3		00:00:15	7 (1)		*TH3*	6	00:16:16	1673 (1)
GnuTLS (3.5.12)	*TH1*	4	00:01:35	570 (1)	Openswan (2.6.50) *	*TH1*	4	00:01:07	378 (1)
	TH2		00:00:06	22 (1)		*TH2*		00:00:04	26 (1)
	TH3		00:00:01	4 (1)		*TH3*		00:00:00	6 (1)
LibreSSL (2.5.4)	*TH1*	4	00:10:27	4008 (1)	PuTTY (0.7)	*TH1*	12	00:03:22	3889 (1)
	TH2		00:01:40	1151 (1)		*TH2*		00:00:07	42 (1)
	TH3	6	00:25:45	1802 (1)		*TH3*		00:00:00	6 (1)
libtomcrypt (1.16)	*TH1*	5	00:01:13	2262 (3)	strongSwan (5.6.3) *	*TH1*	6	00:01:32	2262 (3)
	TH2	16	00:00:11	805 (3)		*TH2*		00:16:36	15747 (3)
	TH3	5	00:04:49	7284 (1)		*TH3*		00:00:24	216 (6)
MatrixSSL (3.9.1) *Certificate*	*TH1*	8	00:01:54	4554 (1)	wolfSSL (3.11.0)	*TH1*	10	00:04:05	14316 (1)
	TH2		00:00:04	202 (1)		*TH2*		00:00:06	26 (1)
	TH3		00:00:22	939 (2)		*TH3*		00:00:00	6 (1)

† Execution Time measured on a commodity laptop with an Intel i7-3740QM CPU and 32GB DDR3 RAM running Ubuntu 16.04.

‡ Shaded cells indicate no discrepancies were found during cross-validation.

* Configured to use their own internal implementations of PKCS#1 v1.5.

5.5 Findings on Recent Implementations

Here we present our findings of testing 15 recent open-source implementations of PKCS#1 v1.5 signature verification. We take the construction-based approach as the golden standard. For each of the test harnesses, while the occurrence of multiple accepting paths signifies problems, it is worth noting that just because an implementation gave only one accepting path does not mean that the implemented verification is robust and correct. In fact, as we show later, some lone accepting paths can still be overly permissive. The summary of results can be found in Table 5.4.

Cross-validation. For performing cross-validation, we use GnuTLS 3.5.12 as our anchor, as it seems to be using a robust construction-based signature verification, and it gave the smallest number of paths with *TH1*. We ran the cross-validation on a commodity laptop with at most 8 query instances in parallel at any time. For each implementation, cross-validating it against the anchor for a particular test harness typically finishes in the scale of minutes. In general, the exact time needed to solve such queries depends on the size and complexity of the constraints, but in this particular context, we have observed that the overall performance is around 1200 queries per every 10 seconds on our commodity laptop.

In the rest of this section, when we show code snippets, block comments with a single star are from the original source code, and those with double stars are our annotations.

Openswan 2.6.50

Openswan is a popular open source IPSec implementation, currently maintained by Xelerance Corporation. Depending on the target platform, Openswan can be configured to use NSS, or its own implementation based on GMP, for managing and processing public-key cryptography. We are particularly interested in testing the latter one.

The verification of PKCS#1 v1.5 RSA signatures in Openswan employs a hybrid approach. Given an O, everything before `AS` is processed by a parser, and then `AS` is checked against some known DER-encoded bytes and the expected $H(m_v)$, which explains why *TH2* and *TH3* both found only a small number of paths, similar to the other hybrid implementations like wolfSSL and BoringSSL. Those paths also successfully cross-validated against the anchor.

Interestingly, despite *TH1* yielding only 1 accepting path, Openswan turns out to have an exploitable vulnerability in its signature verification logic.

Ignoring padding bytes (CVE-2018-15836). As shown in Snippet 5.5.i, during verification, the parser calculates and enforces an expected length of padding. However, while the initial `0x00`, `BT`, and the end of padding `0x00` are verified, the actual padding is simply skipped over by the parser. Since the value of each padding byte is not being checked at all, for a signature verification to succeed, they can take arbitrarily any values. As we will explain later in Section 5.6, this simple but severe oversight can be exploited for a Bleichenbacher-style signature forgery.

Snippet 5.5.i: Padding Bytes skipped in Openswan 2.6.50

```
/* check signature contents */
/* verify padding (not including any DER digest info! */
padlen = sig_len - 3 - hash_len;
... ...
/* skip padding */
if(s[0] != 0x00 || s[1] != 0x01 || s[padlen+2] != 0x00)
  { return "3""SIG padding does not check out"; }
s += padlen + 3;
```

strongSwan 5.6.3

strongSwan is another popular open source IPSec implementation. Similar to Openswan, when it comes to public-key cryptography, strongSwan offers the choice of relying on other cryptographic libraries (*e.g.*, OpenSSL and libgcrypt), or using its own internal implementation, which happens to be also based on GMP. We are focused on testing the latter one. To our surprise, the strongSwan internal implementation of PKCS#1 v1.5 signature verification contains several weaknesses, many of which could be exploited for signature forgery.

Not checking algorithm parameter (CVE-2018-16152). *TH1* revealed that the strongSwan implementation does not reject O with extra garbage bytes hidden in the algorithm parameter, a classical flaw previously also found in other libraries [11,14]. As such, a practical low-exponent signature forgery exploiting those unchecked bytes is possible [7].

Accepting trailing bytes after OID (CVE-2018-16151). *TH2* revealed another exploitable leniency exerted by the parser used by strongSwan during its signature verification. The `asn1_known_oid()` function is used to match a series of parsed OID encoded bytes against known OIDs, but the matching logic is implemented in a way that as soon as a known OID is found to match the prefix of the parsed bytes, it considers the match a success and does not care whether there are remaining bytes in the parsed OID left unconsumed. One can hence hide extra bytes after a correctly encoded OID, and as we will explain in Section 5.6, this can be exploited for a low-exponent signature forgery.

Accepting less than 8 bytes of padding. In fact, strongSwan has another classical flaw. The PKCS#1 v1.5 standard requires the number of padding bytes to be at least 8 [RFC2313, RFC2437]. Unfortunately, during our initial testing with *TH1*, we quickly realized that strongSwan does not check whether `PS` has a minimum length of 8, a flaw previously also found in other implementations [15]. Since `PS` is terminated with `0x00`, during symbolic execution, our initial *TH1* automatically generated test cases where some early byte of `PS` is given the value of `0x00`, and hence the subsequent symbolic bytes would be considered to be part of `AS`. And because strongSwan attempts to parse `AS` using an ASN.1 parser, this resulted in many paths enumerating different possible ASN.1 types with symbolic lengths. After finding this flaw, we have added additional constraints to *TH1* to guide the symbolic execution into not putting `0x00` in `PS`, which in the end resulted in a reasonable number of paths.

Lax ASN.1 length checks. Additionally, the weaknesses regarding algorithm parameter and algorithm OID also led to lenient handling of their corresponding length variables, u and y. This is the reason why *TH3* found several accepting paths, as the parser used during verification enumerated various combinations of values for u and y that it considers acceptable.

axTLS 2.1.3

axTLS is a very small footprint TLS library designed for resource-constrained platforms, which has been deployed in various system on chip (SoC) software stacks, *e.g.*, in Arduino for ESP8266[3], the Light Weight IP stack (LWIP)[4] and MicroPython[5] for various microcontrollers.

Unfortunately, the signature verification in axTLS is some of the laxest among all the recent implementations that we have tested. Its code is aimed primarily at traversing a pointer to the location of the hash value, without enforcing rigid sanity checks on the way. The various weaknesses in its implementation can lead to multiple possible exploits.

Accepting trailing bytes (CVE-2018-16150). We first found that the axTLS implementation accepts O that contains trailing bytes after the hash value, in order words, it does not enforce the requirement on the length of padding bytes, a classical flaw previously found in other implementations [7, 8, 12]. This is also why for both *TH1* and *TH2* there are multiple accepting paths.

Ignoring prefix bytes. On top of that, we found that this implementation also ignores the prefix bytes, including both BT and PB, which also contributes to the various incorrect accepting paths yielded by *TH1* and *TH2*. As shown in Snippet 5.5.ii, this effectively means that the first 10 bytes of O can take arbitrarily any values. Such a logic deviates from what the standard prescribes [RFC2437], and as we will explain later in Section 5.6, an over-permissiveness like this can be exploited to forge signatures when e is small.

Snippet 5.5.ii: Block Type and Padding skipped in axTLS 2.1.3

```
i = 10;/* start at the first possible non-padded byte */
while (block[i++] && i < sig_len);
size = sig_len - i;
```

[3]https://github.com/esp8266/Arduino/tree/master/tools/sdk/lib
[4]https://github.com/attachix/lwirax
[5]https://github.com/micropython/micropython/tree/master/lib

```
/* get only the bit we want */
if (size > 0) {... ...}
```

Ignoring ASN.1 metadata (CVE-2018-16253). Moreover, we found that axTLS does not check the algorithm OID and parameter. In fact, through root-cause analysis, we found that this could be attributed to the parsing code shown in Snippet 5.5.iii below, which skips the entire `AlgorithmIdentifier` part of `AS` (achieved by `asn1_skip_obj()`), until it reaches the hash value (type `OCTET STRING`), making this even laxer than the flaws of not checking algorithm parameter previously found in other libraries [7,11].

Snippet 5.5.iii: Majority of ASN.1 metadata skipped in axTLS 2.1.3

```
if (asn1_next_obj(asn1_sig, &offset, ASN1_SEQUENCE) < 0
 || asn1_skip_obj(asn1_sig, &offset, ASN1_SEQUENCE))
  goto end_get_sig;

if (asn1_sig[offset++] != ASN1_OCTET_STRING)
    goto end_get_sig;
*len = get_asn1_length(asn1_sig, &offset);
ptr = &asn1_sig[offset];           /* all ok */

end_get_sig:
    return ptr;
```

Trusting declared lengths (CVE-2018-16149). Furthermore, using our approach, we have automatically found several test cases that could trigger memory errors at various locations of the axTLS source code. This is because given the various length variables in the ASN.1 structure that are potentially under adversarial control, the parser of axTLS, partly shown in Snippet 5.5.iii, is too trusting in the sense that it uses the declared values directly without sanity checks, so one can put some absurd values in those lengths to try to trick the implementation into reading from illegal memory addresses and potentially crash the program. This is an example of CWE-130 (*Improper Handling of Length Parameter*).

This is also part of the reason why for *TH1*, it took more than 1 hour to finish the execution, as KLEE discovered many test cases that can trick the parsing code into reading z, the ASN.1 length of $H(m_s)$, from some symbolic trailing bytes, which led to several invocations of `malloc()` with huge sizes and hence the long execution time.

MatrixSSL 3.9.1

MatrixSSL requires $|n|$ to be a multiple of 512, so in our test harnesses, we have adjusted the size of the test buffer and padding accordingly. Interestingly, we have observed that MatrixSSL contains 2 somewhat different implementations of PKCS#1 v1.5 signature verification, one for verifying signatures on CRLs, and the other for certificates. Both are using a parsing-based verification approach. Why the two cases do not share the same signature verification function is not clear to us. Nevertheless, we have tested both of them, and to our surprise, one verification is laxer than the other, but both exhibit some forms of weaknesses.

Lax ASN.1 length checks. We first note that for both signature verification functions, their treatments of some of the length variables in `AS` are overly permissive. Quite the opposite of axTLS, we found that MatrixSSL does not fully trust the various ASN.1 lengths, and imposes sanity checks on the length variables. Those, however, are still not strict enough.

For the certificate signature verification, the first 2 ASN.1 lengths variables, w, and x (lengths of the two ASN.1 `SEQUENCE` in `AS`), are allowed ranges of values in the verification. For w, the only checks performed on it are whether it is in the long form, and whether it is longer than the remaining buffer containing the rest of O. Similarly, there exist some sanity checks on x but they are nowhere near an exact match warranted by a construction-based approach. The 2 accepting paths yielded by *TH3* are due to a decision being made on whether x matches exactly the length of the remaining `SEQUENCE` (OID and parameters) that had been consumed, which indicates whether there are extra bytes for algorithm parameters or not. However, this check is done with a macro `psAssert()`, which terminates only if `HALT_ON_PS_ERROR` is defined in the configuration, a flag that is considered to be a debugging option [186], not enabled by default and not recommended for production builds, meaning that many possible values of x, even if they failed the assertion, would still be accepted. When the length of the encoded OID is correct (*i.e.*, 5 for SHA-1), the length of algorithm parameters, y, is not checked at all.

For the CRL signature verification function, the treatments of length variables w, x, and y are also overly permissive, similar to what is done in certificate signature verification. On top of that, the checks on z the declared size of $H(m_s)$ in `AS` is also overly permissive, similar to those on w.

Comparing to a construction-based approach, these implementations are overly permissive and the weaknesses discussed allow some bits in O to take arbitrary any values, which means the verification is not as robust as it ideally should be.

Mishandling Algorithm OID. We found that for the CRL signature verification, there exists another subtle implementation weakness in how it handles the OID of hash algorithms.

As shown in the following snippet, upon finishing parsing the algorithm OID, the verification code would see whether the length of hash output given by the parsed algorithm matches what the caller of the verification function expects. However, since this is again done by the `psAssert()` macro, which as discussed before, does not end the execution with an error code even if the assertion condition fails, and the execution would just fall through. This explains the numerous accepting paths found by *TH2* and *TH3*.

Snippet 5.5.iv: Checking Signature Hash Algorithm in MatrixSSL (CRL)

```
/** outlen := length of H(m) provided by caller,
   oi is the result of OID parsing **/
if (oi == OID_SHA256_ALG)
      {  psAssert(outlen == SHA256_HASH_SIZE);  }
```

```
else if (oi == OID_SHA1_ALG)
     {  psAssert(outlen == SHA1_HASH_SIZE);    }
... ...
else {  psAssert(outlen == SHA512_HASH_SIZE);  }
```

The implications of this flaw is that for the algorithm OID bytes (the length of which is subject to the checks discussed before), they can be arbitrarily any values, since in the end, it is the expected length of $H(m)$ provided by the caller of the verification function that dictates how the rest of the parsing would be performed. Hence the verification is overly permissive and one can get at most 9 arbitrary bytes in the OID part of O this way (*e.g.*, with $H()$ being SHA-256).

Besides, even if `psAssert()` would actually terminate with errors, the above implementation is still not ideal, as the assertion conditions are done based on the length of $H(m)$, not the expected algorithm. We note that the hash size and length of OID are not unique across hash algorithms. Since there are pairs of hash algorithms (*e.g.*, MD5 and MD2; SHA-256 and SHA3-256) such that (1) the length of their OIDs are equal, and (2) the length of their hash outputs are equal, the parser would consider algorithms in each pair to be equivalent, which can still lead to an overly permissive verification. Ideally, this should be done instead by matching the parsed OID against a caller provided expected OID.

GnuTLS 3.5.12

Based on our testing and root-cause analysis, GnuTLS is now using a construction-based approach in its PKCS#1 v1.5 signature verification code, which is a considerable improvement to some of its own vulnerable versions from earlier [7, 14]. This is also reflected in the small number of paths yielded by our test harnesses, even less than those that adopt a hybrid approach. Consequently, we choose this as the anchor for cross-validation.

Dropbear SSH 2017.75

Dropbear implements the SSH protocol, and uses *libtomcrypt* for most of the underlying cryptographic algorithms like the various SHA functions and AES. Interestingly, instead of relying on *libtomcrypt*'s RSA code, for reasons unbeknownst to us, Dropbear SSH has its own RSA implementation, written using the *libtommath* multiple-precision integer library. Based on our root-cause analysis, it appears that the PKCS#1 v1.5 signature verification implemented in the RSA implementation of Dropbear SSH follows the construction-based approach, hence it successfully cross-validated with the anchor and no particular weaknesses were found. In contrast to the bundled *libtomcrypt* which has some signature verification weaknesses (explained below), having its own RSA implemented actually helped Dropbear SSH to avoid some exploitable vulnerabilities.

Comparing to other implementations of construction-based verification (*e.g.*, BoringSSL), the *TH1* of Dropbear SSH took a significantly longer time to run, mainly due to the final comparison after constructing the expected I_v is done in the multiple-precision integer level, not with a typical memory comparison function like `memcmp()`. Nevertheless, it still managed to finish within a reasonable amount of time. As a side benefit, symbolic execution also covered part of the multiple-precision integer *libtommath* code.

libtomcrypt 1.16

Based on our test results, we found that libtomcrypt is also using a parsing-based approach, and its signature verification contains various weaknesses[6].

Accepting trailing bytes. Similar to axTLS, libtomcrypt also has the classical flaw of accepting signatures with trailing bytes after $H(m_s)$, hence a practical

[6]Some of the weaknesses had been independently found by other researchers, leading to certain fixes being introduced in version 1.18.

signature forgery attack is possible when the public exponent is small enough. This is the reason why for *TH1* and *TH2*, there are 3 accepting paths.

Accepting less than 8 bytes of padding. Interestingly, libtomcrypt also has the classical flaw of not checking whether `PS` has a minimum length of 8, similar to strongSwan. Through root-cause analysis, we quickly identified the lax padding check as shown below. Give this verification flaw, to avoid scalability challenges due to symbolic padding bytes, we apply the same workaround to *TH1* as we did for strongSwan.

Snippet 5.5.v: Padding Check in libtomcrypt 1.16

```
for (i = 2; i < modulus_len - 1; i++)
    { if (msg[i] != 0xFF) { break; }  }
/* separator check */
if (msg[i] != 0) {
    /* There was no octet with hexadecimal value
        0x00 to separate ps from m. */
    result = CRYPT_INVALID_PACKET;
    goto bail;
}
/** ... start ASN.1 parsing at msg[i+1] ... **/
```

Lax AlgorithmIdentifier length check. Furthermore, despite the fact that *TH3* yielded only one accepting path, it turns out there is another subtle weakness in libtomcrypt. We found that in `AS`, the length x of `AlgorithmIdentifier` (the inner ASN.1 `SEQUENCE`) is checked only loosely, despite the constraints imposed on x by the verification code. This is because the constraints are mostly simple sanity and boundary checks such that x cannot be too small or too large, but the x is not required to match exactly to a concrete value (*i.e.*, 9 with explicit `NULL` parameter and $H()$ being SHA-1). This is partly because the ASN.1 parser used by libtomcrypt, *re-encodes* the bytes of an ASN.1 simple type that were just parsed, to calculate the actual length that was consumed. Hence, when given a child of ASN.1 `OID`, the length of the parent `SEQUENCE`, as in the case of `AlgorithmIdentifier`, was not checked strictly. This is also why for *TH2* it needed to change a handful of lines more, to workaround the re-encoding of OID which has decisions to be made for each byte, depending on whether it is less than 128 (short form) or not (long form).

Because the verification code would accept a range of values for x, this gives some bits in the middle of `AS` that one can choose arbitrary and is hence overly permissive.

mbedTLS 2.4.2

Based on the results of our testing, mbedTLS appears to be also using the parsing-based verification approach. The relatively larger number of paths from *TH1* and *TH3* can be attributed to the underlying ASN.1 parser, as there are various decisions (*e.g.*, whether the lengths are in the long form or not) to be made during parsing. We note that despite each of {*TH1, TH2, TH3*} gave exactly one accepting path, only the paths extracted by *TH1* and *TH2* were successfully cross-validated with the other implementations. Upon close inspection of the one and only accepting path yielded by *TH3*, we realized it contains a subtle verification weakness, which was also caught by cross-validation.

Lax algorithm parameter length check.

Interestingly, in mbedTLS 2.4.2, the checks imposed on y, the length of algorithm parameter, are in fact too lenient. Through root-cause analysis with CPT, we found that the only constraints imposed came from the parser, as shown in Snippet 5.5.vi. There are 2 constraints, one is whether the most significant bit is on, which the parser uses to decide how it should obtain the actual length. The other one is whether the declared length is longer than what is remaining in the buffer.

Snippet 5.5.vi: Only parsing and sanity checks imposed on y in mbedTLS 2.4.2

```
if( ( **p & 0x80 ) == 0 ) *len = *(*p)++;
else { ... ... }

if( *len > (size_t) ( end - *p ) )
```

```
        return( MBEDTLS_ERR_ASN1_OUT_OF_DATA );
```

Since after the parser consumed y, there would be 22 bytes left in the buffer (assuming no parameter bytes, $2+20$ for a SHA-1 hash), it turns out the verification code would accept any values of y not larger than 22, which allows some bits of `AS` to be arbitrarily chosen and is hence overly permissive.

BoringSSL 3112, BearSSL 0.4 and wolfSSL 3.11.0

BoringSSL is a fork of OpenSSL, refactored and maintained by Google. We found its PKCS#1 v1.5 signature verification uses a hybrid approach. Everything before `AS` in O is handled and checked by a parser that scans through the buffer, and then `AS` is copied out. The verification code then constructs its own expected version of $\texttt{AS}_v$ using $H(m_v)$ and some hard-coded ASN.1 prefixes, and then compares $\texttt{AS}_v$ against `AS`. This observed behavior is consistent with what was reported earlier [11]. Consequently, the total number of paths are reasonably small, with each of {*TH1, TH2, TH3*} yielding exactly one accepting path. BearSSL and wolfSSL both behaved quite similar to BoringSSL, and all 3 implementations successfully cross-validate against the anchor with no discrepancies observed. wolfSSL yielded more paths in *TH1* due to a slightly different handling of `PB`, and BearSSL yielded more paths in *TH2* due to extra handling of the case of absent parameter.

OpenSSL 1.0.2l and LibreSSL 2.5.4

We found that OpenSSL adopts a parsing-based verification approach, which partly explains why some higher number of paths were yielded by *TH2* and *TH3*. The slightly longer execution time of *TH3* can partly be attributed to the concretization workaround. Despite these, no verification weaknesses were found in this recent version of OpenSSL, which is perhaps unsurprising given that it had gone through years of scrutiny by security researchers [7]. LibreSSL is a fork of OpenSSL maintained by the OpenBSD community since 2014 after the infamous Heartbleed vulnerability.

The two are actually quite similar when it comes to PKCS#1 v1.5 signature verification, both using a similar parsing-based approach and the test harnesses all yielded comparable numbers of execution paths.

PuTTY 0.7

We found that the PuTTY implementation of PKCS#1 v1.5 signature verification is highly reminiscent of a construction-based approach. The left-most 2 bytes of O containing `0x00` and `BT` are checked first, followed by a check on `PB` with an expected length (which depends on $|n|$), and then `AS` before $H(m_s)$ is checked against some hard-coded ASN.1 encoded bytes, and finally, $H(m_s)$ is checked. Cross-validation found no discrepancies and no signature verification weaknesses were detected.

Interestingly, even after sufficient rejection criteria has been hit (*e.g.*, `BT` is not `0x01`), the verification continues with other checks, until all has been finished and then an error would finally be returned. Since the later checks before the verification function returns do not alter a rejection return code back into an acceptance, this is not a verification weakness. We suspect this insistence on traversing the whole buffer containing O might be an attempt to avoid timing side channels.

However, as explained below with Example 2, such an implementation presents a small hurdle for symbolic execution, as the number of paths due to `if` statements (the series of checks) exhibits a multiplicative build-up, leading to a scalability challenge observed in our first round experiment with *TH1*. Consequently, we modified the source to adopt an '*early return*' logic, like a typical implementation of `memcmp()` would do. That is, once a sufficient rejection condition has been reached, the verification function returns with an error without continuing with further checks, so that the number of paths would build up additively. This explains why the number of lines changed in PuTTY is slightly higher than the others.

```
if (symBuf[0] != 0) ret = 0;
if (symBuf[1] != 1) ret = 0;
if (symBuf[2] != 2) ret = 0;
return ret;
```

```
if (symBuf[0] != 0) return 0;
if (symBuf[1] != 1) return 0;
if (symBuf[2] != 2) return 0;
```

Example 2: For number of execution paths, the snippet on right builds up additively, but the one on left does so multiplicatively.

OpenSSH 7.7

OpenSSH is another open source SSH software suite. For handling PKCS#1 v1.5 signatures, it relies on OpenSSL (calling `RSA_public_decrypt()`) to perform the RSA computation and process the paddings of O. Afterwards, it compares the `AS` returned by OpenSSL against its constructed version, hence it is somewhat of a hybrid approach. Cross-validation found no discrepancies and no weaknesses were detected in the verification.

Interestingly, instead of simply using `memcmp()`, the comparison against the constructed `AS` is done using a custom constant time comparison, as shown below:

```
/** p1,2 point to buffers of equal size(=n) **/
for (; n > 0; n--) ret |= *p1++ ^ *p2++;
return (ret != 0);
```

This explains why *TH3* found in total only 2 paths of relatively larger constraints, as such a timing safe comparison would aggregate (with `OR`) the comparison (with `XOR`) of each byte in the two buffers. Semantically, the 2 execution paths mean either all length variables u, w, x, y, z in *TH3* match their expected values exactly, or at least one of them does not.

5.6 Exploiting Our New Findings

Here we discuss how to exploit the several weaknesses presented in the previous section. For ease of discussion, we focus on SHA-1 hashes, but the attacks can be adapted to handle other hash algorithms by adjusting the lengths of appropriate components. Though low-exponent RSA public keys are rarely seen in the Web PKI nowadays [187], there are specific settings where low-exponent keys are desired (*e.g.*, with extremely resource-constrained devices). Historically, a small public exponent of $e = 3$ has been recommended for better performance [RFC3110], and there are key generation programs that still mandate small public exponents [181].

Signature forgery against Openswan

The flaw of *ignoring padding bytes* effectively means Openswan would accept a malformed O' in the form of

`0x00 || 0x01 || GARBAGE || 0x00 || AS` ,

which can be abused in a manner similar to the signature forgery attack exploiting the weakness of not checking algorithm parameters found in some other implementations as discussed in previous work [7].

This has serious security implications. We note that in the context of IPSec, the key generation program `ipsec_rsasigkey` forces $e = 3$ without options for choosing larger public exponents [181]. Since the vulnerable signature verification routine is used by Openswan to handle the `AUTH` payload, the ability to forge signatures might enable man-in-the-middle adversaries to spoof an identity and threaten the authentication guarantees delivered by the `IKE_AUTH` exchange when RSA signature is used for authentication.

Given the implementation flaw allows for certain bytes *in the middle* of O' to take arbitrarily any values, the goal of the attack is to forge a signature $S' = (k_1 + k_2)$, such that when the verifier computes $O' = S'^3 = (k_1 + k_2)^3 = {k_1}^3 + 3{k_1}^2k_2 + 3{k_2}^2k_1 + {k_2}^3$, the following properties would hold:

1. the *most significant bits* of ${k_1}^3$ would be those that need to be matched exactly *before* the unchecked padding bytes, which is simply `(0x00 || 0x01)`;

2. the *least significant bits* of ${k_2}^3$ would become those that need to be matched exactly *after* the unchecked padding bytes, which is simply `(0x00 || AS)`;

3. the *most significant bits* of ${k_2}^3$ and the *least significant bits* of ${k_1}^3$, along with $3{k_1}^2k_2 + 3{k_2}^2k_1$, would stay in the unchecked padding bytes.

One influential factor to the success of such attack is whether there are enough unchecked bytes for an attacker to use. An insufficient amount would have the terms of expanding $(k_1 + k_2)^3$ overlapping with each other, make it difficult for the three properties to hold. However, since the flaw we are exploiting is on the handling of padding bytes, the number of which grows linearly with $|n|$, assuming the same public exponent, a longer modulus would actually contribute to the attacker's advantage and make it easier to forge a signature. Specifically, assuming SHA-1 hashes and $e = 3$, given $|n| \geq 1024$ bits, it should be easy to find k_1 and k_2 that satisfy the three properties without worrying about overlaps.

Finding k_1. The main intuition used is that a feasible k_1 can be found by taking a cubic root over the desired portion of O'. For instance, in the case of $|n| = 1024$ bits, `0x00 || 0x01 || 0x00 ... 0x00` is simply 2^{1008} (with 15 zero bits in front), hence a simple cubic root would yield a $k_1 = 2^{336}$.

In the more general cases where $|n| - 15 - 1$ is not a multiple of 3, the trailing garbage could be used to hide an over-approximation. One can first compute $t_1 = \lceil \sqrt[3]{2^{|n|-15-1}} \rceil$ and then sequentially search for the largest possible r such that $((t_1/2^r + 1) \cdot 2^r)^3$ gives `0x00 || 0x01 || GARBAGE`. Then k_1 would be $(t_1/2^r + 1) \cdot 2^r$. This is to make as many ending bits of k_1 to be zero as possible, to avoid overlapping terms in the expansion of $(k_1 + k_2)^3$. For example, when $|n| = 2048$ bits, we found $r = 676$ bits and $k_1 = 3 \cdot 2^{676}$.

Finding k_2. The intuition is that to get `(0x00 || AS)` with ${k_2}^3$, the modular exponentiation can be seen as computed over a much smaller n'' instead of the full modulus n. While finding $\phi(n)$ reduces to factorizing n, which is believed to be impractical when n is large, finding $\phi(n'')$ can be quite easy.

One can consider $S'' =$`(0x00 || AS)` and $n'' = 2^{|S''|}$, where $|S''|$ is the size of `AS` in number of bits plus 8 bits for the end of padding `0x00`.

Now k_2 has to satisfy ${k_2}^e \equiv S'' \pmod{n''}$. Since n'' is a power of 2, we can guarantee k_2 and n' are coprime by choosing an odd numbered S'' with a fitting hash value. Also, $\phi(n'') = \phi(2^{|S''|}) = 2^{|S''|-1}$.

One can then use the Extended Euclidean Algorithm to find f such that $ef \equiv 1 \pmod{2^{|S''|-1}}$. With f found, k_2 would simply be ${S''}^f \pmod{n''}$.

We have implemented attack scripts assuming $e = 3$ and SHA-1 hashes, and were able to forge signatures that would be successfully verified by Openswan 2.6.50 given any $|n| = 1024$ and $|n| = 2048$ moduli.

Signature forgery (1) against strongSwan

The flaw of *not checking algorithm parameter* can be directly exploited for signature forgery following the algorithm given in [7] (which is very similar to the attack we described previously against Openswan). Assuming $e = 3$, $|n| = 1024$ bits and SHA-1 hashes, the expected iterations required to brute-force a fake signature is reported to be 2^{21} [7].

Signature forgery (2) against strongSwan

Likewise, the flaw of *accepting trailing bytes after OID* can be exploited following the steps used in the forgery attack against Openswan as described before, by adjusting what ${k_1}^3$ and ${k_2}^3$ represent. Under the same parameter settings, it should require a comparable number of iterations as *signature forgery (1)* does discussed above.

Signature forgery (3) against strongSwan

Interestingly, the flaw of *accepting less than 8 bytes of padding* can be exploited together with the algorithm parameter flaw to make it easier to forge signatures. In fact, the two flaws together means such an O' with no paddings at all would be accepted:

```
/** all numbers below are hexadecimals **/
00 01 00 30 7B 30 63 06   05 2B 0E 03 02 1A 05 5A
GARBAGE  04 16 SHA-1(m')
```

The length of algorithm parameter `0x5A` is calculated based on $|n|$ (in this case 1024 bits) and the size of hash. Then by simply adjusting what ${k_1}^3$ and ${k_2}^3$ represent in the attack against Openswan, given $e = 3$ and $|n| \geq 1024$ bits, the forgery will easily succeed. We implemented this new variant of attack and confirmed that the fake signatures generated actually work.

Signature forgery (4) against strongSwan

Similarly, the forgery attack exploiting trailing bytes after OID could also benefit from the absence of padding, as an O' like the followings would be accepted by strongSwan:

```
/** all numbers below are hexadecimals **/
00 01 00 30 7B 30 63 06   5F 2B 0E 03 02 1A
GARBAGE 05 00 04 16 SHA-1(m')
```

The length of algorithm OID `0x5F` is calculated based on $|n|$ (in this case 1024 bits) and the size of hash. The attack against Openswan would work here as well, simply by adjusting what ${k_1}^3$ and ${k_2}^3$ represent. Signature forgery would again easily succeed given $e = 3$ and $|n| \geq 1024$ bits. We have also implemented this new attack variant and confirmed that the fake signatures generated indeed work.

Signature forgery (1) against axTLS

Given that there exist performance incentives in using small exponents with the kinds of resource-constrained platforms that axTLS targets, a practical signature forgery attack as described in [7] could be made possible by the flaw of *accepting trailing bytes*. Specifically, when $|n| = 1024$, assuming $e = 3$ and SHA-1 hashes, the expected number of trials before a successful forgery is reported to be around 2^{17} iterations, which takes only several minutes on a commodity laptop [7]. As a larger $|n|$ would allow for more trailing bytes, hash algorithms that yield longer hashes could be attacked similarly, *e.g.*, assuming $e = 3$ and SHA-256 hashes, a modulus with $|n| = 2048$ bit should easily yield a successful forgery. Similarly, such an attack would also work against a larger public exponent with an accordingly longer modulus.

Signature forgery (2) against axTLS

Separately, the weakness of *ignoring ASN.1 metadata* as shown in Snippet 5.5.iii, can also be exploited for a low-exponent signature forgery. Due to the majority of `AS` being skipped over, axTLS would accept an O' like this:

```
/** all numbers below are hexadecimals **/
00 01 FF FF FF FF FF FF   FF FF 00 30 5D 30 5B
GARBAGE 04 16 SHA-1(m')
```

where the lengths `0x5D` and `0x5B` are calculated based on $|n|$ and size of hash to make sure the skipping would happen correctly. Then the forgery attack against Openswan described before can be easily adapted to work here by adjusting what ${k_1}^3$ and ${k_2}^3$ represent. Given $|n| \geq 1024$, forgery should easily succeed. We have tested the adapted attack script and the forged signatures it generates indeed worked on axTLS.

Signature forgery (3) against axTLS

Knowing that axTLS also *ignores prefix bytes* as shown in Snippet 5.5.ii, the *signature forgery (1)* described above which exploits *unchecked trailing bytes* can be made even easier to succeed, by making the first 11 bytes all 0 (including the end of padding indicator). Adapting the analysis from previous work [7], the signature value O is essentially a number less than 2^{935} (assuming $|n| = 1024$, the first 88 bits are all zeros, with 2 additional zero bits from the first `0x30` byte of `AS`). The distance between two consecutive perfect cubes in this range is

$$\begin{aligned} k^3 - (k-1)^3 = 3k^2 - 3k + 1 &< 3 \cdot 2^{624} - 3 \cdot 2^{312} + 1 \\ &< 2^{626} \qquad\qquad (\because k^3 < 2^{935}) \end{aligned} \tag{5.1}$$

which is less than the 656 bits that an attacker can choose arbitrarily (46 bytes are fixed, due to the 35-byte `AS` containing a desired SHA-1 hash and the 11 bytes in front), so a signature forgery should easily succeed, by preparing an attack input O' containing hash of an attacker-chosen m', and the attack signature S' can be found by simply taking the cubic root of O'. Once the verifier running axTLS 2.1.3 received S', it would compute $O' := S'^3 \mod n$, and despite O' being malformed, the verification would go through.

Signature forgery (4) against axTLS

Furthermore, the weakness of *ignoring ASN.1 metadata*, can be exploited together with the previous attack, to make the signature forgery even easier. The intuition is that, knowing the parsing code would skip over the ASN.1 prefix (the two `0x30` ASN.1 `SEQUENCE`) according to the length declared, an attacker can spend the minimal number of bytes on `AS` to keep the parser entertained, with an O' like this:

```
/** all numbers below are hexadecimals **/
00 00 00 00 00 00 00 00   00 00 00 30 00 30 00 04
H().size   H(m')   TRAILING
```

and spend the gained extra free bytes at the end as trailing ones. While for SHA-256 and $|n| = 1024$, a signature forgery attack exploiting *only* trailing bytes has the expected iterations of about 2^{145} [7], however, if we use this joint attack strategy instead, this bound can be pushed down much lower and the attack becomes practical. Specifically, assuming SHA-256, the joint attack strategy would have $11+6+32 = 49$ bytes fixed, and 79 trailing bytes (632 bits) at the end that the attacker can choose arbitrarily, more than the bound of 626 bits on the distance between two perfect cubes from eq. (5.1), so a forgery should easily succeed by taking the cubic root as described before. We have implemented attack scripts and successfully performed this new variant of signature forgery on axTLS 2.1.3 with $e = 3, |n| = 1024$ and for both SHA-1 and SHA-256.

Denial of Service against axTLS

We further note that because of the *trusting* nature of the parser in axTLS, an illegal memory access attack against axTLS with absurd length values is also possible, which might crash the verifier and result in a loss of service. Specifically, following the previous forgery attack, we prepared an attack script that generates signatures which would yield a z (the length of hash) of `0x84`, and the illegal memory access induced by this absurd value had successfully crashed the verifier in our experiments.

We further note that such a denial of service attack can be even easier to mount than a signature forgery in the context of certificate chain verification. This is due to the fact that axTLS verifies certificate chains in a bottom-up manner, which contributes to an attacker's advantage: even if low-exponent public keys are rarely used by real CAs in the wild, to crash a verifier running axTLS, one can purposefully introduce a counterfeit intermediate CA certificate that uses a low-exponent as the j-th one in the chain, and forge a signature containing absurd length values as described above and put it on the $(j + 1)$-th certificate. Due to the bottom-up verification, before the code traverses up the chain and attempts to verify the j-th counterfeit certificate against the $(j-1)$-th one, it would have already processed the malicious signature on the $(j+1)$-th certificate and performed some illegal memory access. While a bottom-up certificate chain verification is not inherently wrong, but because of the weaknesses in the signature verification, the bottom-up design has an unexpected exploitable side effect. **This highlights why a signature verification code needs to be robust regardless of the choice of e.**

Signature forgery (1) against libtomcrypt

Just like the flaw of *accepting trailing bytes* in axTLS, the same flaw in libtomcrypt 1.16 can also be exploited in a signature forgery attack if the e is small enough and $|n|$ is large enough, following the same attack algorithm described in [7].

Signature forgery (2) against libtomcrypt

We note that the flaw of *accepting less than* 8 *bytes of padding* found in libtomcrypt 1.16 also has serious security implications. Combining this with the attack exploiting trailing bytes, the low-exponent signature forgery can be made even easier. Specifically, an attacker can craft an O' like this:

```
/** all numbers below are hexadecimals **/
00 01 00 || AS || TRAILING || EXTRA TRAILING
```

The intuition behind is that one can shorten the padding as much as possible, and spend the extra bytes at the end. Assuming $|n| = 1024, e = 3$ and $H()$ is SHA-1, this attack has 38 bytes fixed, and hence $1024 - 38 \cdot 8 = 720$ bits that the attacker can choose arbitrarily. Since in this case, O' is essentially a number $< 2^{1010}$, the distance between two consecutive perfect cubes in this range is

$$\begin{aligned} k^3 - (k-1)^3 = 3k^2 - 3k + 1 &< 3 \cdot 2^{674} - 3 \cdot 2^{337} + 1 \\ &< 2^{676} \qquad\qquad (\because k^3 < 2^{1010}), \end{aligned}$$

which is less than the 720 bits that can be chosen arbitrarily, so a signature forgery would succeed easily. We have implemented an attack script and verified the success of such a signature forgery attack against libtomcrypt 1.16.

Other weaknesses

We note that not all the weaknesses found can immediately lead to a practical Bleichenbacher-style low-exponent signature forgery attack. For example, even though the other weaknesses in mbedTLS 2.4.2, MatrixSSL 3.9.1 and libtomcrypt 1.16 regarding *lax length variable checks* allow for some bits to take arbitrary any values, given that the number of free bits gained due to those weaknesses appear to be somewhat limited, it is not immediately clear how to exploit them for signature forgery. Nevertheless, those implementations are accepting signatures that should otherwise be rejected, which is less than ideal and might potentially be taken advantage of when combined with some other unexpected vulnerabilities in a specific context.

5.7 Disclosure and Fixes

In an effort of responsible disclosure, we have notified vendors of the weak implementations so that they can have their signature verifications hardened. CVEs are requested and assigned selectively on the basis that a weakness can lead to immediate practical attacks as outlined above. Developers of *MatrixSSL* have acknowledged and confirmed our findings, and have released fixes. *strongSwan* has fixed the problems since version 5.7.0 and released further patches for older versions. *Openswan* has fixed the exploitable weakness since their 2.6.50.1 release and incorporated one of our forged signatures into their unit tests. *libtomcrypt* developers have created a ticket regarding the parser weakness and are currently investigating it. We developed a patch for *axTLS* and tested it with our approach before releasing it, and our patch has been incorporated by the axTLS ESP8266 port as well as the upstream axTLS maintainer. At the time of writing, we are awaiting responses from the vendor of *mbedTLS*.

5.8 Conclusion

In this chapter, we propose to enhance symbolic execution with meta-level search and constraint provenance tracking, for automatically generating concolic test cases and easier root-cause analysis. As a demonstration, we analyzed `15` open-source implementations of PKCS#1 v1.5 signature verification and found semantic flaws in 6 of them. We also discuss in details how to exploit some of the flaws for signature forgery. We have publicly released the relevant source code and artifacts like extracted path constraints, so other researchers and practitioners can reproduce our work and leverage it to test other implementations.

In the long run, perhaps it is worth reconsidering the design of incorporating a flexible but complex structure inside security-critical objects like digital signatures. While an ASN.1 DER structure like `AS` is highly extensible and can easily accommodate new hash algorithms, the reality is, new standardized algorithms seldom get introduced, and complicating a common but critical routine that gets invoked multiple times daily for a flexibility that is enjoyed only once in a while might not seem to be worthwhile.

6. SUMMARY

Security mechanisms are critical for protecting communications and other digital assets in a potentially hostile networked environment. Network connection makes it much easier and more convenient to share information and resources, but also opens up the possibilities of various remote attacks, which is exactly why it is critical to correctly deploy proper security mechanisms. In this thesis, taking a *top-down* approach, we study the problem of improper security mechanism deployment, and help improve the understanding of how to design, implement and analyze such mechanisms through a series of systematic evaluations.

First, we begin at the application layer, and study the problem of deploying content distribution apps on mobile platforms. As smartphones and other portable devices are becoming ubiquitous, digital multimedia contents are increasingly being consumed on such platforms. We define a hierarchy of possible adversaries and attack surfaces, and demonstrate how due to unjustified trust assumptions, weak design patterns, as well as flawed enforcement of control policies and best practices, many content distribution apps fail to adequately protect the digital contents that they deliver, leading to potential loss of revenue for the corresponding businesses. To educate and help the community avoid similar deployment pitfalls, we dissect the flaws found in the vulnerable apps and map the relevant patterns to CWEs. We have also responsibly discussed our findings with the companies that are affected.

Then we look at the problem of validating X.509 certificates, a critical step in common deployments of TLS, the de facto standard for encrypting Internet traffic. X.509 certificates are widely used in TLS as a means for achieving authentication. The security guarantees of TLS deployments often hinge on the assumption that implementations are correctly validating X.509 certificates, and flawed certificate validation can lead to loss of service and impersonation attacks. Previous state of the art relies on unguided black-box fuzzing to test implementations of X.509 certificate validation. Focusing on small-footprint TLS libraries targeting embedded and IoT systems, we propose a principled approach that leverages symbolic execution to achieve better coverage and uncover hidden logical flaws buried deep in the code. We found that when it comes to certificate validation, many TLS libraries deviate from what the specification prescribes, and our approach is able to find more subtle problems comparing to previous work. We have communicated our findings with the library maintainers and most of the issues are already fixed. 3 new CVEs of medium severity have been assigned to the new flaws that we found.

Finally we study the problem of PKCS#1 v1.5 signature verification, a critical algorithm that is often used in X.509 certificate validation, as well as in some other network protocols like SSH, IPSec and DNSSEC. Previous research based on manual analysis has found that flawed implementations of PKCS#1 v1.5 can be susceptible to signature forgery attacks. Using the PKCS#1 v1.5 signature verification as another case study, we revamp the subject of analyzing semantic correctness with symbolic execution, which is a theme also shared by our previous work on X.509 certificates. Semantic correctness is interesting as a research topic, particularly for security-critical mechanisms, because many existing work on software testing focus on finding memory errors, which can be easily prevented with memory-safe programming languages, but not so much for logical flaws. We discuss how concolic test cases can be automatically generated by exploiting the structural features of the PKCS#1 v1.5 signature scheme, and we introduce a novel constraint provenance tracking mechanism that helps to identify the line of code that imposes incorrect path constraints. To our surprise, we found hidden flaws that enable new variants of the Bleichenbacher '06 signature forgery attacks in several crypto libraries and IPSec software, which could be used to break authentication under certain configurations. 6 new CVEs (3 medium and 3 high severity) have been assigned to these implementation flaws. We have helped the library developers fix their signature verification code and released the artifacts we used in this research.

Future work. It would be interesting to see how the analysis of semantic correctness can be extended to investigate implementations of other stateful network protocols. For example, recently it has been found that there exist a state machine transition bug in `libssh`[1], which allows a trivial authentication bypass.

Beyond the work presented in this thesis, another aspect of security mechanism deployments that is also worth inspecting is configuration issues. Most of this thesis is devoted to the study of design and implementation of mechanisms, but bad configurations (*e.g.*, allowing the use of broken ciphers) can also negatively impact security in actual deployments. It remains to be seen how to automatically infer configurations from deployments of security mechanisms in a non-intrusive manner.

[1] `https://arstechnica.com/information-technology/2018/10/bug-in-libssh-makes-it-amazingly-easy-for-hackers-to-gain-root-access/`

REFERENCES

REFERENCES

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

[2] Y. Chen and Z. Su, "Guided differential testing of certificate validation in ssl/tls implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ACM, 2015, pp. 793–804.

[3] L. Valenta, N. Sullivan, A. Sanso, and N. Heninger, "In search of curveswap: Measuring elliptic curve implementations in the wild," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, 2018, pp. 384–398.

[4] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices." in *USENIX Security Symposium*, 2012.

[5] M. Hastings, J. Fried, and N. Heninger, "Weak keys remain widespread in network devices," in *Proceedings of the 2016 Internet Measurement Conference.* ACM, 2016, pp. 49–63.

[6] L. Zhang, D. Choffnes, D. Levin, T. Dumitras, A. Mislove, A. Schulman, and C. Wilson, "Analysis of SSL certificate reissues and revocations in the wake of Heartbleed," in *the 2014 ACM IMC*, pp. 489–502.

[7] U. Kühn, A. Pyshkin, E. Tews, and R. Weinmann, "Variants of bleichenbacher's low-exponent attack on PKCS#1 RSA signatures," in *Sicherheit 2008: Sicherheit, Schutz und Zuverlässigkeit. Konferenzband der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 2.-4. April 2008 im Saarbrücker Schloss.*, 2008, pp. 97–109.

[8] H. Finney, *Bleichenbacher's RSA signature forgery based on implementation error*, 2006 (accessed Jul 06, 2018), https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html.

[9] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, "SymCerts: Practical Symbolic Execution For Exposing Noncompliance in X. 509 Certificate Validation Implementations," in *IEEE Symposium on Security and Privacy*, 2017, pp. 503–520.

[10] C. Chen, C. Tian, Z. Duan, and L. Zhao, "Rfc-directed differential testing of certificate validation in ssl/tls implementations," in *Proceedings of the 40th International Conference on Software Engineering.* ACM, 2018, pp. 859–870.

[11] Bugzilla, *RSA PKCS#1 signature verification forgery is possible due to too-permissive SignatureAlgorithm parameter parsing*, 2014 (accessed Jul 18, 2018), https://bugzilla.mozilla.org/show_bug.cgi?id=1064636.

[12] Common Vulnerabilities and Exposures, *CVE-2006-4340*, 2006 (accessed Jul 18, 2018), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4340.

[13] ——, *CVE-2006-4790*, 2006 (accessed Aug 01, 2018), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4790.

[14] S. Josefsson, *[gnutls-dev] Original analysis of signature forgery problem*, 2006 (accessed Jul 21, 2018), https://lists.gnupg.org/pipermail/gnutls-dev/2006-September/001240.html.

[15] Intel Security: Advanced Threat Research, *BERserk Vulnerability – Part 2: Certificate Forgery in Mozilla NSS*, 2014 (accessed Jul 06, 2018), https://bugzilla.mozilla.org/attachment.cgi?id=8499825.

[16] H. Böck, J. Somorovsky, and C. Young, "Return of bleichenbacher's oracle threat (ROBOT)," in *27th USENIX Security Symposium*, 2018.

[17] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson, "Analyzing Forged SSL Certificates in the Wild," in *Security and Privacy (SP), 2014 IEEE Symposium on.* IEEE, 2014, pp. 83–97.

[18] X. d. C. de Carnavalet and M. Mannan, "Killed by proxy: Analyzing client-end tls interception software," in *Network and Distributed System Security Symposium*, 2016.

[19] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson, "The security impact of https interception," in *Network and Distributed System Security Symposium*, 2017.

[20] L. Waked, M. Mannan, and A. Youssef, "To intercept or not to intercept: Analyzing tls interception in network appliances," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018.

[21] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell, "Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 223–238.

[22] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 2013, pp. 445–459.

[23] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan, "FLEXTLS: A Tool for Testing TLS Implementations," in *Proceedings of the 9th USENIX Conference on Offensive Technologies*, ser. WOOT'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831211.2831212

[24] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "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.* ACM, 2012, pp. 38–49.

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

[26] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting ssl usage in applications with sslint," in *2015 IEEE Symposium on Security and Privacy.* IEEE, 2015, pp. 519–534.

[27] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing api usages through semantic cross-checking," in *25th USENIX Security Symposium (USENIX Security 16).* Austin, TX: USENIX Association, Aug. 2016, pp. 363–378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun

[28] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. R. Butler, and A. Alkhelaifi, "Securing ssl certificate verification through dynamic linking," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2014, pp. 394–405.

[29] B. Reaves, N. Scaife, A. M. Bates, P. Traynor, and K. R. Butler, "Mo (bile) Money, Mo (bile) Problems: Analysis of Branchless Banking Applications in the Developing World." in *USENIX Security Symposium*, 2015, pp. 17–32.

[30] C. Zuo, Q. Zhao, and Z. Lin, "Authscope: Towards automatic discovery of vulnerable authorizations in online services," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2017, pp. 799–813.

[31] C. Zuo, W. Wang, Z. Lin, and R. Wang, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services." in *Network and Distributed System Security Symposium*, 2016.

[32] B. Mueller and S. Schleier, "Owasp mobile app security requirements and verification v 1.0," 2018. [Online]. Available: https://github.com/OWASP/owasp-masvs/releases/download/1.0/OWASP_Mobile_AppSec_Verification_Standard_v1.0.pdf

[33] A. Blaich and A. Striegel, "Is high definition a natural drm?" in *Proceedings of 18th International Conference on Computer Communications and Networks*, Aug 2009, pp. 1–4.

[34] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *TISSEC*, vol. 12, no. 2, 2008.

[35] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008, pp. 209–224.

[36] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *ICSE*, 2011, pp. 1066–1071.

[37] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[38] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *ESEC/FSE*. ACM, 2005.

[39] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," in *PLDI*, 2002.

[40] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[41] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, 2012.

[42] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*. ACM, 2005.

[43] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin, "Automating software testing using program analysis," *Software, IEEE*, vol. 25, no. 5, pp. 30–37, 2008.

[44] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 669–685. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032360

[45] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[46] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein, "Analyzing protocol implementations for interoperability," in *NSDI*, 2015.

[47] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *NSDI*, 2012.

[48] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A soft way for openflow switch interoperability testing," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 265–276. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413207

[49] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 361–377. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815422

[50] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified nat," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 141–154.

[51] S. Chaki and A. Datta, "Aspier: An automated framework for verifying security protocol implementations," in *2009 22nd IEEE Computer Security Foundations Symposium*, July 2009, pp. 172–185.

[52] M. Aizatulin, F. Dupressoir, A. D. Gordon, and J. Jürjens, "Verifying Cryptographic Code in C: Some Experience and the Csec Challenge," in *Formal Aspects of Security and Trust*, 2012.

[53] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution," in *ACM CCS '11*, 2011, pp. 331–340.

[54] R. Corin and A. Manzano, "Efficient symbolic execution for analysing cryptographic protocol implementations," in *Proceedings of the Third International Conference on Engineering Secure Software and Systems*, ser. ESSoS'11, 2011, pp. 58–72.

[55] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *IEEE Symposium on Security and Privacy*, 2015.

[56] J. De Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 193–206.

[57] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the 2016 ACM CCS*, 2016, pp. 1492–1504.

[58] R. B. Evans and A. Savoia, "Differential testing: A new approach to change detection," in *ESEC-FSE companion '07*, 2007, pp. 549–552.

[59] D. J. Bernstein, "Cache-timing attacks on aes," 2005 (accessed Sep 19, 2018). [Online]. Available: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[60] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2006, pp. 201–215.

[61] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "Stealing keys from pcs using a radio: Cheap electromagnetic attacks on windowed exponentiation," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2015, pp. 207–228.

[62] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[63] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "Ecdsa key extraction from mobile devices via nonintrusive physical side channels," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1626–1638.

[64] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *International cryptology conference*. Springer, 2014, pp. 444–461.

[65] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2017.

[66] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *11th USENIX Workshop on Offensive Technologies*, 2017.

[67] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1," in *Annual International Cryptology Conference*. Springer, 1998, pp. 1–12.

[68] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay, "Efficient padding oracle attacks on cryptographic hardware," in *Advances in Cryptology – CRYPTO 2012*, 2012, pp. 608–625.

[69] S. Vaudenay, "Security flaws induced by cbc paddingapplications to ssl, ipsec, wtls..." in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2002, pp. 534–545.

[70] B. Möller, T. Duong, and K. Kotowicz, "This poodle bites: Exploiting the ssl 3.0 fallback," 2014.

[71] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: Breaking the tls and dtls record protocols," in *2013 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2013, pp. 526–540.

[72] E. Ronen, K. G. Paterson, and A. Shamir, "Pseudo constant time implementations of tls are only pseudo secure."

[73] C. Hlauschek, M. Gruber, F. Fankhauser, and C. Schanes, "Prying open Pandora's box: KCI attacks against TLS," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[74] C. Garman, M. Green, G. Kaptchuk, I. Miers, and M. Rushanan, "Dancing on the lip of the volcano: Chosen ciphertext attacks on apple imessage," in *25th USENIX Security Symposium 2016*. USENIX Association, pp. 655–672.

[75] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper, "Breaking LTE on layer two," in *IEEE Symposium on Security & Privacy (SP)*. IEEE, May 2019.

[76] A. Biryukov, G. Leurent, and A. Roy, "Cryptanalysis of the "kindle" cipher," in *International Conference on Selected Areas in Cryptography*. Springer, 2012, pp. 86–103.

[77] S. Crosby, I. Goldberg, R. Johnson, D. Song, and D. Wagner, "A cryptanalysis of the high-bandwidth digital content protection system," in *ACM Workshop on Digital Rights Management*. Springer, 2001, pp. 192–200.

[78] R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services," in *USENIX Security Symposium*, 2013, pp. 687–702.

[79] doridori, *Android-Security-Reference/TEE.md*, 2017 (accessed Feb 06, 2018), https://github.com/doridori/Android-Security-Reference/blob/master/hardware/TEE.md.

[80] Android Developers, *Saving Files - Choose Internal or External Storage*, (accessed Feb 02, 2018), https://developer.android.com/training/data-storage/files.html#InternalVsExternalStorage.

[81] Editors of Inc., “MAZ Systems - New York, NY,” 2017. [Online]. Available: https://www.inc.com/profile/maz-systems

[82] MITRE Corporation, *CVE-2018-5686*, 2018 (accessed Feb 05, 2018), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5686.

[83] Synk Security Team, “Zip Slip Vulnerability,” 2018. [Online]. Available: https://github.com/snyk/zip-slip-vulnerability

[84] Microsoft Corporation, *PlayReady Header Specification*, 2017 (accessed Feb 07, 2018), https://www.microsoft.com/playready/documents/.

[85] ——, *Microsoft PlayReady Content Protection Technology*, 2015 (accessed Feb 07, 2018), https://www.microsoft.com/playready/documents/.

[86] *DRM License cacheable*, 2010 (accessed Feb 07, 2018), https://social.msdn.microsoft.com/Forums/silverlight/en-US/b0220e8a-0660-49aa-8353-18d12ae285dd/drm-license-cacheable.

[87] mediaarcadmin, *ios_stack/Sources/Classes/DRM/drmenvelope.h*, 2010 (accessed Jan 29, 2018), https://github.com/mediaarcadmin/ios_stack/blob/master/Sources/Classes/DRM/drmenvelope.h.

[88] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest we remember: cold boot attacks on encryption keys,” in *17th USENIX Security Symposium*, 2008, pp. 45–60.

[89] *DMCA security research exemption for consumer devices*, 2016 (accessed Feb 07, 2018), https://www.ftc.gov/news-events/blogs/techftc/2016/10/dmca-security-research-exemption-consumer-devices.

[90] David Ruddock, *[Weekend Poll] Is Your Primary Android Device Rooted?*, 2014 (accessed Feb 06, 2018), http://www.androidpolice.com/2014/11/23/weekend-poll-is-your-primary-android-device-rooted-2/.

[91] Andy Boxall, *80% of Android phone owners in China have rooted their device*, 2015 (accessed Feb 06, 2018), http://www.businessofapps.com/80-android-phone-owners-china-rooted-device/.

[92] Magisk Manager, *Download Magisk Manager Latest Version 5.5.5 For Android 2018*, 2018 (accessed Feb 06, 2018), https://magiskmanager.com/#Magisk_Root_Universal_Systemless_Interface.

[93] Chainfire, *[CENTRAL] CF-Auto-Root*, 2012 (accessed Feb 07, 2018), https://forum.xda-developers.com/showthread.php?t=1980683.

[94] *SuperSU - Android Apps on Google Play*, 2017 (accessed Feb 07, 2018), https://play.google.com/store/apps/details?id=eu.chainfire.supersu.

[95] Rohit Salecha, *Pentesting Android Apps Using Frida*, 2017 (accessed Feb 06, 2018), https://www.notsosecure.com/pentesting-android-apps-using-frida/.

[96] J. W. Matt Joseph and contributors, “Rootcloak,” 2016. [Online]. Available: https://github.com/devadvance/rootcloak

[97] Ryne Hager, *Google may have updated SafetyNet detection, breaking some root hiding*, 2017 (accessed Feb 06, 2018), http://www.androidpolice.com/2017/06/14/google-may-updated-safetynet-detection-breaking-root-hiding-methods/.

[98] ——, *SafetyNet can detect Magisk again, but a fix is in the works*, 2017 (accessed Feb 06, 2018), http://www.androidpolice.com/2017/07/16/safetynet-can-detect-magisk-fix-works/.

[99] Collin Mulliner and John Kozyrakis, *Inside Android's SafetyNet Attestation*, 2017 (accessed Feb 06, 2018), https://www.blackhat.com/docs/eu-17/materials/eu-17-Mulliner-Inside-Androids-SafetyNet-Attestation.pdf.

[100] Nahuel Cayetano Riva, *Anti-instrumentation techniques: I know you're there, Frida!*, 2015 (accessed Feb 06, 2018), https://crackinglandia.wordpress.com/2015/11/10/anti-instrumentation-techniques-i-know-youre-there-frida/.

[101] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *IEEE Symposium on Security and Privacy*, 2017.

[102] F. Falcon and N. Riva, "Dynamic binary instrumentation frameworks: I know you're there spying on me," in *Reverse Engineering Conference*, 2012.

[103] X. Li and K. Li, "Defeating the transparency features of dynamic binary instrumentation. blackhat us," 2014.

[104] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *26th USENIX Security Symposium*, 2017, pp. 33–49.

[105] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 289–298.

[106] L. Boney, A. H. Tewfik, and K. N. Hamdy, "Digital watermarks for audio signals," in *Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on*. IEEE, 1996, pp. 473–480.

[107] J. A. Bloom and C. Polyzois, "Watermarking to track motion picture theft," in *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, vol. 1. IEEE, 2004, pp. 363–367.

[108] C. Grothoff, K. Grothoff, R. Stutsman, L. Alkhutova, and M. Atallah, "Translation-based steganography," *Journal of Computer Security*, vol. 17, no. 3, pp. 269–303, 2009.

[109] M. Topkara, U. Topkara, and M. J. Atallah, "Information hiding through errors: a confusing approach," in *Security, Steganography, and Watermarking of Multimedia Contents IX*, vol. 6505. International Society for Optics and Photonics, 2007.

[110] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[111] ITU-T Recommendation X.509 (2005) — ISO/IEC 9594-8:2005, "Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks," *International Telecommunication Union*, 2005.

[112] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," Internet Requests for Comments, Tech. Rep. 5280, May 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[113] "CVE-2016-1115," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1115.

[114] "CVE-2016-5669," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-5669.

[115] "CVE-2016-5672," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-5672.

[116] "CVE-2016-2180," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2180.

[117] "CVE-2016-5655," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-5655.

[118] "CVE-2016-3664," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-3664.

[119] "CVE-2016-2113," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2113.

[120] "CVE-2016-1563," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-1563.

[121] "CVE-2016-2562," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2562.

[122] "CVE-2016-2047," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2047.

[123] "CVE-2015-5655," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-5655.

[124] "CVE-2014-0092," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0092.

[125] "CVE-2014-1266," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266.

[126] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 2, no. 42, pp. 230–265, 1936.

[127] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, M. Roberts, S. Setty, and B. Zill, "Ironfleet: Proving practical distributed systems correct," in *SOSP*, 2015.

[128] K. L. McMillan, *Symbolic model checking.* Springer, 1993.

[129] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, “Automatic predicate abstraction of c programs,” in *ACM SIGPLAN Notices*, vol. 36, no. 5. ACM, 2001, pp. 203–213.

[130] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, “Zing: A model checker for concurrent software,” in *CAV*, 2004.

[131] G. Holzmann and M. Smith, “A practical method for verifying event-driven software,” in *ICSE.* ACM, 1999.

[132] R. Jhala and R. Majumdar, “Software model checking,” *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.

[133] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy abstraction,” in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 58–70.

[134] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *CAV*, 2000, pp. 154–169.

[135] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, “Synergy: A new algorithm for property checking,” in *FSE*, 2006.

[136] T. Ball and S. Rajamani, “The slam project: Debugging system software via static analysis,” *SIGPLAN Not.*, vol. 37, no. 1, 2002.

[137] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, “The software model checker blast: Applications to software engineering,” *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5, 2007.

[138] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, “Modular verification of software components in C,” *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 388–402, 2004.

[139] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, “Satabs: Sat-based predicate abstraction for ansi-c,” in *TACAS.* Springer-Verlag, 2005.

[140] J. Esparza, S. Kiefer, and S. Schwoon, “Abstraction refinement with Craig interpolation and symbolic pushdown systems,” in *TACAS.* Springer, 2006.

[141] S. Löwe, “Cpachecker with explicit-value analysis based on cegar and interpolation,” in *TACAS.* Springer-Verlag, 2013.

[142] G. Brat, K. Havelund, S. Park, and W. Visser, “Model checking programs,” in *IEEE International Conference on Automated Software Engineering (ASE).* Citeseer, 2000, pp. 3–12.

[143] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* New York, NY, USA: Cambridge University Press, 1993.

[144] J. Jaffar, V. Murali, J. Navas, and A. Santosa, “Tracer: A symbolic execution tool for verification,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, P. Madhusudan and S. Seshia, Eds. Springer Berlin Heidelberg, 2012, vol. 7358, pp. 758–766.

[145] E. M. Clarke, S. Jha, and W. Marrero, "Verifying security protocols with brutus," *TOSEM*, vol. 9, no. 4, 2000.

[146] P. Godefroid, "Model checking for programming languages using verisoft," in *POPL*. ACM, 1997, pp. 174–186.

[147] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[148] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *ACM PLDI '12*, pp. 193–204.

[149] *HTTPS client is here for the Photon!*, 2015 (accessed Nov 3, 2016), https://community.particle.io/t/https-client-is-here-for-the-photon-by-the-glowfi-sh-team/15934.

[150] *spark / firmware / communication / lib*, 2016 (accessed Nov 3, 2016), https://github.com/spark/firmware/tree/master/communication/lib.

[151] *Arduino/libraries/ESP8266WiFi/src/include/ssl.h*, 2017 (accessed Aug 2, 2018), https://github.com/esp8266/Arduino/blob/master/libraries/ESP8266WiFi/src/include/ssl.h.

[152] *micropython/extmod*, 2017 (accessed Feb 2, 2017), https://github.com/micropython/micropython/tree/master/extmod.

[153] "CVE-2016-6303," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-6303.

[154] "CVE-2016-7052," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-7052.

[155] "CVE-2016-6305," https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-6305.

[156] P.-H. Kamp, "Please put openssl out of its misery," *Queue*, vol. 12, no. 3, pp. 20:20–20:23, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602649.2602816

[157] *how to install curl and libcurl*, (accessed Nov 3, 2016), https://curl.haxx.se/docs/install.html.

[158] *ipsvd - internet protocol service daemons - installation*, (accessed Feb 2, 2017), http://smarden.org/ipsvd/install.html.

[159] *Package: gatling (0.12cvs20120114-4) high performance web server and file server*, (accessed Feb 2, 2017), https://packages.debian.org/wheezy/gatling.

[160] *A Python library that encapsulates wolfSSL's wolfCrypt API.*, (accessed Feb 2, 2017), https://pypi.python.org/pypi/wolfcrypt/0.2.0.

[161] *mbed TLS (PolarSSL) wrapper*, (accessed Feb 2, 2017), https://pypi.python.org/pypi/python-mbedtls/0.6.

[162] W. M. McKeeman, "Differential testing for software," *DIGITAL TECHNICAL JOURNAL*, vol. 10, no. 1, pp. 100–107, 1998.

[163] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, in *Handbook of Satisfiability*, ch. Satisfiability Modulo Theories.

[164] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 519–531.

[165] C. Adams and S. Lloyd, *Understanding PKI: Concepts, Standards, and Deployment Considerations*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[166] G. Nelson and D. C. Oppen, "Fast decision procedures based on congruence closure," *J. ACM*, vol. 27, no. 2, pp. 356–364, Apr. 1980.

[167] S. Legg, "ASN.1 Module Definition for the LDAP and X.500 Component Matching Rules," RFC 3727, Mar. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc3727.txt

[168] C. Gardiner and C. Wallace, "ASN.1 Translation," RFC 6025, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc6025.txt

[169] "ASN.1 JavaScript decoder," https://lapo.it/asn1js/.

[170] *wolfSSL ChangeLog*, 2016 (accessed Oct 22, 2016), https://www.wolfssl.com/wolfSSL/Docs-wolfssl-changelog.html.

[171] *mbed TLS 2.2.0, 2.1.3, 1.3.15 and PolarSSL 1.2.18 released*, 2015 (accessed Mar 14, 2017), https://tls.mbed.org/tech-updates/releases/mbedtls-2.2.0-2.1.3-1.3.15-and-polarssl.1.2.18-released.

[172] *Certificate Template Extensions: Application Policy*, (accessed Oct 28, 2016), https://technet.microsoft.com/en-us/library/cc731792(v=ws.11).aspx.

[173] C. Young, *Flawed MatrixSSL Code Highlights Need for Better IoT Update Practices*, 2016 (accessed Feb 6, 2017), http://www.tripwire.com/state-of-security/security-data-protection/cyber-security/flawed-matrixssl-code-highlights-need-for-better-iot-update-practices/.

[174] *IPv6 Ready Logo Program*, 2016 (accessed Sept 04, 2016), https://www.ipv6ready.org.

[175] *High Definition Logos*, 2016 (accessed Sept 04, 2016), http://www.digitaleurope.org/Services/High-Definition-Logos.

[176] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, *A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions.* Cham: Springer International Publishing, 2014, pp. 646–662. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08867-9_43

[177] *Vulnerability Note VU#396440 - MatrixSSL contains multiple vulnerabilities*, 2016 (accessed Feb 6, 2017), http://www.kb.cert.org/vuls/id/396440.

[178] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the tls 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy (SP).* IEEE, 2017, pp. 483–502.

[179] S. Gao, H. Chen, and L. Fan, "Padding Oracle Attack on PKCS#1 V1.5: Can Non-standard Implementation Act As a Shelter?" in *Proceedings of the 12th International Conference on Cryptology and Network Security - Volume 8257*, 2013, pp. 39–56.

[180] J. Galea, S. Heelan, D. Neville, and D. Kroening, "Evaluating manual intervention to address the challenges of bug finding with KLEE," *CoRR*, vol. abs/1805.03450, 2018. [Online]. Available: http://arxiv.org/abs/1805.03450

[181] *Ubuntu Manpage: ipsec_rsasigkey - generate RSA signature key*, 2018 (accessed Aug 21, 2018), http://manpages.ubuntu.com/manpages/bionic/man8/ipsec_rsasigkey.8.html.

[182] "OSDI'08 Coreutils Experiments," http://klee.github.io/docs/coreutils-experiments.

[183] "Tutorial on How to Use KLEE to Test GNU Coreutils," https://klee.github.io/tutorials/testing-coreutils.

[184] *OpenSSL Security Advisory [5th September 2006]*, 2006 (accessed Aug 02, 2018), https://www.openssl.org/news/secadv/20060905.txt.

[185] ITU-T, *Information technology ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)* , 2002 (accessed Aug 01, 2018), https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf.

[186] INSIDE Secure, *MatrixSSL Developers Guide*, 2017 (accessed Jul 21, 2018), https://github.com/matrixssl/matrixssl/blob/master/doc/matrixssl_dev_guide.md#debug-configuration.

[187] A. Delignat-Lavaud, M. Abadi, A. Birrell, I. Mironov, T. Wobber, and Y. Xie, "Web PKI: Closing the Gap between Guidelines and Practices." in *NDSS*, 2014.

VITA

VITA

Sze Yiu Chau received his BSc in Information Technology with First-class honours from Department of Computing, The Hong Kong Polytechnic University in 2013. Born and raised in Hong Kong, he speaks fluent Cantonese, Mandarin Chinese (Putonghua) and English. He started pursuing his Ph.D. in the Department of Computer Science at Purdue University from Fall 2013. He worked under the supervision of Dr. Ninghui Li and Dr. Aniket Kate. Prior to joining Purdue, he was an engineering trainee at EMPA (Eidgenössische Materialprüfungs- und Forschungsanstalt), the Swiss Federal Laboratories for Material Science and Technology during Fall 2011 and Spring 2012. He also visited Thayer School of Engineering at Dartmouth on an undergraduate research program in Summer 2012. His research interest broadly lies in the area of network and system security, software testing and verification techniques, automated reasoning, and privacy over the Internet.