Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Plugging in Trust and Privacy

–

Three Systems to Improve Widely Used Ecosystems

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von
Sebastian Rainer Gerling

Saarbrücken,
September 2014

Tag des Kolloquiums: 16. Dezember 2014

Dekan: Prof. Dr. Markus Bläser

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Matteo Maffei
Berichterstattende:	Prof. Dr. Michael Backes
	Prof. Dr. Andreas Zeller
Akademischer Mitarbeiter:	Dr. Marcel Böhme

To Sarah.

Zusammenfassung

Heutige Mobilgeräte mit Touchscreen haben unsere Kommunikationsgewohnheiten grundlegend geändert. Ihre intuitive Benutzbarkeit gepaart mit unbegrenztem Internetzugang erlaubt es uns jederzeit und überall zu kommunizieren und führt dazu, dass immer mehr (vertrauliche) Informationen publiziert werden. Des Weiteren hat der Erfolg mobiler Geräte zur Einführung neuer Dienste die auf vertraulichen Daten aufbauen (z.B. positionsabhängige Dienste) beigetragen. Mit den aktuellen Mobilgeräten wurde zudem das Internet die wichtigste Informationsquelle (z.B. für Nachrichten) und die Nutzer müssen sich auf die Korrektheit der von dort bezogenen Daten verlassen. Allerdings bieten die involvierten Systeme weder robuste Datenschutzgarantien, noch die Möglichkeit die Korrektheit bezogener Daten zu verifizieren.

Diese Dissertation führt drei neue Mechanismen für das Vertrauen und den Datenschutz ein, die die aktuelle Situation in weit verbreiteten Systemen verbessern. WebTrust, ein robustes Authentizitäts- und Integritätssystem ermöglicht es den Nutzern sowohl die Korrektheit als auch die Autorenschaft von über HTTP übertragenen Daten zu verifizieren. X-pire! und X-pire 2.0 bieten ein digitales Ablaufdatum für Bilder in sozialen Netzwerken um Daten auch nach der Publikation noch vor Zugriff durch Dritte zu schützen. AppGuard ermöglicht das Durchsetzen von feingranularen Datenschutzrichtlinien für Drittanbieteranwendungen in Android um einen angemessen Schutz der Nutzerdaten zu gewährleisten.

Abstract

The era of touch-enabled mobile devices has fundamentally changed our communication habits. Their high usability and unlimited data plans provide the means to communicate any place, any time and lead people to publish more and more (sensitive) information. Moreover, the success of mobile devices also led to the introduction of new functionality that crucially relies on sensitive data (e.g., location-based services). With our today's mobile devices, the Internet has become the prime source for information (e.g., news) and people need to rely on the correctness of information provided on the Internet. However, most of the involved systems are neither prepared to provide robust privacy guarantees for the users, nor do they provide users with the means to verify and trust in delivered content.

This dissertation introduces three novel trust and privacy mechanisms that overcome the current situation by improving widely used ecosystems. With WebTrust we introduce a robust authenticity and integrity framework that provides users with the means to verify both the correctness and authorship of data transmitted via HTTP. X-pire! and X-pire 2.0 offer a digital expiration date for images in social networks to enforce post-publication privacy. AppGuard enables the enforcement of fine-grained privacy policies on third-party applications in Android to protect the users privacy.

Background of this Dissertation

This dissertation is based on the papers mentioned in the following. The author contributed to all papers as one of the main authors. The actual implementation of the first version of X-pire! [P1] was mainly done by Stefan Lorenz and Julian Backes, whereas the author contributed the majority of the conceptual design of X-pire! [P1]. For X-pire 2.0 [P8], the author contributed the whole system design, major parts of the implementation as well as the evaluation, while the JPEG embedding (incl. the implementation) was adapted from X-pire!. A discussion of the central idea behind X-pire! is provided in [P9]. In WebTrust [P7], the author contributed to major parts of the overall system design. Moreover, the author was responsible for the client-side design, the client-side and partially also the server-side implementation (except for the VDS library) as well as parts of the evaluation. AppGuard [P4, P3, P10, P5, P2, P6] was developed together with Philipp von Styp-Rekowsky. He designed and implemented the actual enforcement mechanism based on inline reference monitoring, whereas the author developed the policy specification language EXSPoX. Furthermore, the author was responsible for choosing suitable use-cases and for evaluating the chosen use-cases. This included the specification of the corresponding security and privacy policies. The author supervised the bachelor thesis by Gregor Geßner [T2] who implemented the automatic transformation of policies written in the EXSPoX policy language into the actual JAVA-based policies for AppGuard. In addition, the author supervised the master thesis of Erik Derr [T1] who developed, based on the author's idea, a powerful static analysis framework with a particular focus on outgoing Internet connections called Bati. The analysis framework provided valuable information regarding the behavior of apps and builds, thereby, the future basis for actual privacy policies in AppGuard.

Besides the mentioned papers and supervised theses closely related to this dissertation, the author was further involved in the work on the Android Security Framework [S3, S4] and Scippa [S1]. In addition, the author supervised the master theses of Liviu Teris [T4] and of Sven Obser [T3]. Liviu Teris designed and implemented, based on the author's idea, an interface for secure data containers in Android based on state-of-the art trusted computing components. Sven Obser designed and implemented, based on the author's idea, a firewall mechanism to prevent data leaks detected by the analysis framework Bati from Erik Derr.

[P1] J. Backes, M. Backes, M. Dürmuth, S. Gerling, and S. Lorenz. "X-pire! – A digital expiration date for images in social networks." In: *CoRR* abs/1112.2649 (2011).

[P2] M. Backes, S. Gerling, and P. von Styp-Rekowsky. "Gezielte Vergabe von App-Rechten in Android – Smartphones für den Arbeitsalltag sichern." In: *IT-Sicherheit Ausgabe* 1/2013 (2013).

[P3] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. "AppGuard – Enforcing User Requirements on Android Apps." In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 543–548.

[P4] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. "AppGuard – Fine-grained Policy Enforcement for Untrusted Android Applications." In: *Proceedings of the 8th International Workshop on Data Privacy Management (DPM 2013)*. Vol. 8247. Lecture Notes in Computer Science. Springer, 2014, pp. 213–231.

[P5] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. *AppGuard - Fine-grained Policy Enforcement for Untrusted Android Applications*. Tech. rep. A/02/2013. Saarland University, 2013.

[P6] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. *AppGuard - Real-time policy enforcement for third-party applications*. Tech. rep. A/02/2012. Saarland University, 2012.

[P7] M. Backes, R. Gerling, S. Gerling, S. Nürnberger, D. Schröder, and M. Simkin. "WebTrust – A Comprehensive Authenticity and Integrity Framework for HTTP." In: *Proceedings of the 12th International Conference on Applied Cryptography and Network Security (ACNS 2014)*. Vol. 8479. Lecture Notes in Computer Science. Springer, 2014, pp. 401–418.

[P8] M. Backes, S. Gerling, S. Lorenz, and S. Lukas. "X-pire 2.0 – A User-Controlled Expiration Date and Copy Protection Mechanism." In: *Proceedings of the 29th ACM Symposium on Applied Computing (SAC 2014)*. ACM, 2014.

[P9] S. Gerling and R. W. Gerling. "Wie realistisch ist ein "Recht auf Vergessenwerden"?" In: *Datenschutz und Datensicherheit* 37.7 (2013), pp. 445–446.

[P10] P. von Styp-Rekowsky, S. Gerling, M. Backes, and C. Hammer. "Callee-site Rewriting of Sealed System Libraries." In: *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS'13)*. Vol. 7781. Lecture Notes in Computer Science. Springer, 2013, pp. 33–41.

Other Papers of the Author

[S1] M. Backes, S. Bugiel, and S. Gerling. "Scippa: System-Centric IPC Provenance Provisioning on Android." In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[S2] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder. "Acoustic Side-Channel Attacks on Printers." In: *Proceedings of the 19th Usenix Security Symposium*. Usenix Association, 2010, pp. 307–322.

[S3] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. *Android Security Framework: Enabling Generic and Extensible Access Control on Android*. Tech. rep. A/01/2014. Saarland University, 2014.

[S4] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. "Android Security Framework: Extensible Multi-Layered Access Control on Android." In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[S5] S. Bugiel, E. Derr, S. Gerling, and C. Hammer. “Advances in Mobile Security.” In: *Proceedings of the 8th Security Research Conference on Future Security*. Fraunhofer Verlag, 2013.

Supervised Theses

[T1] E. Derr. “Verifying the Internet Access of Android Applications.” M.Sc. thesis. Saarland University, 2011.

[T2] G. Geßner. “AppGuard: Automatic Compilation of Security Specifications to Java Code.” B.Sc. thesis. Saarland University, 2013.

[T3] S. Obser. “User-controlled Internet Connections in Android.” M.Sc. thesis. Saarland University, 2011.

[T4] L. Teris. “Securing User-data in Android: A conceptual approach for consumer and enterprise usage.” M.Sc. thesis. Saarland University, 2012.

Acknowledgments

First of all, I am deeply indebted to my Bachelor, Master, and PhD advisor Michael Backes for his excellent support and guidance during the last seven years. It was an honor to work with him; he provided me with an excellent education during my studies and had always an open ear for questions. His excitement and enthusiasm for IT security and privacy and his motivation created an inspiring working atmosphere and made it great fun to work with him. Besides Michael Backes, I would like to say special thanks to Bettina Balthasar and Joachim Lutz for spending endless hours during the last years to make the Information Security and Cryptography Group home to so many students.

I want to thank Christian Hammer, Matteo Maffei, and Dominique Schröder for many fruitful discussions, valuable feedback, and enthusiastic work in our joint research. In addition, I would like to thank Andreas Zeller for agreeing to review this thesis. Furthermore, I thank Gerhard Weikum and Meinard Müller for their support and advice during my PhD studies.

Many thanks also go to my research collaborators Julian Backes, Sven Bugiel, Erik Derr, Markus Dürmuth, Stefan Lorenz, Stephan Lukas, Stefan Nürnberger, Mark Simkin, and Philipp von Styp-Rekowsky for their excitement and their contributions to our joint research. Additionally, I would like to thank all my colleagues at the Information Security and Cryptography Group and CISPA. In particular, I want to mention my room mates Boris Köpf and Oana Ciobotaru, as well as our tennis crew Milivoj Simeonovski, Kim Pecina, Manuel Reinert, and Hazem Torfah. In addition, I would like to explicitly thank Esfandiar Mohammadi, Raphael Reischuk, Fabienne Eigner, and Kim Pecina for taking this exciting PhD journey with me, and Fabian Bendun as well as Sebastian Meiser for many interesting discussions.

I thank the International Max Planck Research School for Computer Science for funding the first three years of my PhD studies and its staff team as well as the staff team of the Saarbrücken Graduate School of Computer Science for their continuous assistance.

I am grateful to my parents and family for their constant support. A big “thank you” also goes to my friends for their patience and understanding during the last years when my time for them was rare.

Finally, I would like to thank my beloved wife Jenny for her great support and for always believing in me. This thesis would not have been possible without her. She always ensured that I had the time needed to finish my work by taking care of everything else. I dedicate this dissertation to my daughter Sarah and thank her for the great patience she has had with her father since she was born. I am sorry for one of her first words being “Papa Arbeit”.

Contents

List of Figures

List of Tables

1 Introduction

Digital communication has fundamentally changed our way of living. Within the last two decades, online services such as Web sites, e-mail, online social networks, or blogging services have started to change communication habits of people all over the world, and this happened for many good reasons: these services provide high usability combined with much better and faster connections to people, especially if the communication partners are spread around the globe. But the utility of recent services also goes far beyond. People started to publish and share lots of personal information on the Internet such as photos, videos, or comments as well as opinions. The whole social interaction is enriched with digital media on the Internet. The pervasive spreading of mobile devices featuring a permanent Internet connection provides the means for our new communication channels. Mobile online connections lately also reached our personal infrastructure, such as cars and our homes to provide comfortable remote control features. We simply start to connect basically everything in our surrounding; any place, any time.

However, these new technologies have also several downsides: most people communicate with each other using plain text messages, which is especially problematic for confidential data. In addition, the majority verifies communication partners solely based on pseudonyms or e-mail addresses. But how do they know that they are indeed communicating with the intended recipient (authenticity)? It is fundamental to all our communication that we can reliably ensure that we are indeed communicating with the intended people. In this dissertation, we present WebTrust [P7], the first framework to reliably prove authorship and integrity of transmitted Web resources. It seamlessly integrates into the existing infrastructure and can be perfectly intertwined with HTTPS to additionally provide both authenticated connections to the delivering server and data confidentiality during transmission.

Whenever we publish sensitive information such as, for example, personal comments, blog posts, or, potentially unflattering, pictures on the Internet (for example, via social

networks) we loose full control of our data. Even if we restrict the access to a small subset of people, how can we ensure that our images are not further copied outside this group? Current technologies do not provide any useful mechanism to protect our data once it is published while still allowing to share it. For our personal data it is essential that we can publish information without loosing control and knowledge about its spreading. Only if we can enforce our control, we are able to efficiently provide post-publication privacy.

In this dissertation, we present X-pire! [P1], the first mechanism that provides a digital expiration date for images in social networks. To achieve this, images are encrypted and embedded into valid JPEG-images that are later uploaded to social networks. After the images are published, access to these images is controlled by controlling access to the decryption key. Once the expiration date is reached, the decryption key can no longer be accessed. The major challenge of this approach was to reliably embed encryptions into JPEG-files such that they survive JPEG recompression in upload routines of social networks. X-pire! constitutes the technical limit that can be achieved with a pure software-based approach. Its security is based on the major assumption that users of the system are considered trustworthy. This assumption is removed in the follow-up version X-pire 2.0 [P8], which is presented thereafter. X-pire 2.0 is the first solution that also solves the data duplication problem: no user, even if malicious, is able to create copies of keys or decrypted images at any point in time. The system leverages trusted computing components to ensure full control over digital data in the post-publication phase.

But our privacy is not only at risk when we explicitly publish our data. Today, the majority of people accesses online services from mobile devices that consolidate a huge set of functionality. Typical features include, but are not limited to, normal communication services (telephone, voice over IP, text messages, e-mail), location-based services (GPS, navigation, POI search), multimedia (photos, videos, and audio) and recording (microphone and photo/video camera), as well as office, business and Internet applications. No other class of devices has ever provided more features in a single device. The security concepts and solutions in place on mobile devices, however, did not keep pace with the evolution of mobile devices. Even worse, well known solutions such as virus scanners or effective firewalls as known from desktop computers did not directly find the way to mobile devices. Moreover, with the deployment of modern mobile devices the whole software installation process was subject to a paradigm shift: applications, so called apps, are installed from a central application store. This led to a high diversity of apps with highly specialized functionality. Typical developers are no longer big companies, but commonly unknown developers that are often lacking the expertise and experience of a professional and secure software development. The installation of these potentially unprofessionally developed apps together with intentionally malicious apps can lead to huge privacy and security risks: users of Android, for example, can either decide to trust an application, or they cannot install it at all. We overcome this situation for the Android operating system by introducing AppGuard [P6, P5, P2, P10, P3, P4]. AppGuard provides to possibility to enforce fine-grained security and privacy policies on untrusted third-party applications. In this dissertation, we present our new policy specification language EXSPoX, how it can be used to specify policies

for AppGuard, and how these policies provide great utility in a variety of use-cases.

1.1 Contributions

This dissertation introduces WebTrust, X-pire!/X-pire 2.0, and AppGuard to improve both trust and privacy in our today's workflow with modern mobile devices.

1.1.1 WebTrust

WebTrust leverages state-of-the-art cryptographic primitives to provide a comprehensive authenticity and integrity framework for HTTP-based communication. The system establishes trust both in the correctness and the authorship of transmitted documents and forms the basis for our omnipresent usage of Web services as prime source of information. WebTrust is designed to support mobile devices as they play a central role in our today's communication habits and integrates seamlessly into the current infrastructure.

1.1.2 X-pire! and X-pire 2.0

The X-pire!-tools are first to provide a digital expiration date for images in existing social networks. Both X-pire! and X-pire 2.0 rely on a novel technique to embed encrypted data into JPEG-images such that they survive the recompression in upload routines of social networks. X-pire! provides a solution purely in software and poses trust assumptions on users and the operating system whereas X-pire 2.0 removes these assumptions by leveraging state of the art trusted computing components to provide strong security guarantees.

1.1.3 AppGuard

AppGuard provides a novel approach to inline reference monitoring that overcomes Android's deficiencies with its permission system and allows users to enforce fine-grained security and privacy policies without requiring modifications to the Android operating system, root privileges, or the like. The system integrates a high-level policy language and supports the full instrumentation process of applications on standard Android smartphones and tablet computers. We prove the utility of our approach by applying AppGuard to several real-world Android third-party applications to protect the user's privacy. Additionally, we show how AppGuard can even be used to mitigate vulnerabilities in third-party applications and the operating system.

1.2 Outline

This dissertation is organized as follows: In Section 2 we introduce our comprehensive authenticity and integrity framework called WebTrust. Afterwards, we introduce X-pire! and X-pire 2.0 in Section 3. Finally, Section 4 introduces AppGuard with its policy language EXSPoX before we conclude this dissertation in Section 5.

2 WebTrust

Today, the vast majority of our daily communication relies on Internet technologies. The HTTP-protocol [73] has become the standard for requesting and delivering static, dynamic, and even real-time streamed Web content from Web servers. But the HTTP protocol was merely designed for transferring data between a server and a client without considering the security of this connection. The HTTP protocol does not provide any state information as it would be required for individual content per user (e.g., after logging in to an account) and it does not provide any security guarantees: in particular, it neither provides the integrity or authenticity of documents, nor does it provide data confidentiality or the like. In this section, we introduce with WebTrust a comprehensive authenticity and integrity framework that adds the missing authenticity and integrity features. The section is based on [P7].

2.1 Motivation

The de-facto standard on the Internet for achieving a secured HTTP communication is provided by the HTTP over TLS protocol, which is commonly referred to as HTTPS [186]. But HTTPS was designed to provide a (mutually) authenticated and secure connection between a client and a server: both the client and the server identity can be checked based on their public certificates[1] and the transmission of data is encrypted and provides integrity protection. But if an attacker manages to compromise an account on the Web server or the Web server itself to manipulate the stored Web resources, no client is able to detect these manipulations, neither with HTTP nor with HTTPS. Moreover, in many cases only a provable authorship and end-to-end integrity is required, and not data confidentiality. When we look at today's applications on the Internet and consider how

[1] Ideally, both the client and the server certificates are signed by members of a trusted PKI to facilitate their verification.

often people already rely on unprotected information on the Internet, this situation is completely unsatisfactory. If we consider, for example, a typical Web 2.0 blogging Web site such as Twitter [227], users want to rely on information that is provided by other users. And people are used to trust such information without further considerations, which has already led to severe problems: In 2013, attackers hacked into the Twitter account of associated press and published the following text [141]:

Breaking: Two Explosions in the White House and Barack Obama is injured

This message destroyed a market value of about 136 billion dollars [141]. And our current Web standards provide no means to prevent such attacks. Both the HTTP and the HTTPS protocol were simply not designed to prove to clients the authenticity of documents with respect to their authors, or to later prove the correctness of such documents to third parties. HTTP does not provide any integrity guarantees and HTTPS can only guarantee the correct transmission of documents between clients and servers. So all existing standards were not designed to provide the desired features. In addition, all closely related systems that were previously proposed but are not yet widely used, do either not cover all relevant use-cases, or they do not provide all features desired for a comprehensive authenticity and integrity framework. We provide a feature comparison of all closely related approaches as well as in-depth discussions of these approaches in Section 2.11.

2.2 Problem Description

The intuitive approach to solve the lack of authenticity and integrity of documents for HTTP would be to sign every single document before the Web server delivers it. But this would still not provide authenticity and integrity of a document transferred from its author to a client. It would merely protect the transmission between the Web server and the client. This could be solved by moving already the signing operation to the author who signs documents right after their creation. Although this would solve the challenge of providing integrity and authenticity for static documents, this would leave the problem unsolved for dynamically created content as well as for real-time streamed content. The challenge is to achieve authenticity and integrity with respect to the author of static, dynamic, and real-time streamed content. The challenge is, furthermore, to enable the proof of authenticity and integrity of documents in front of third parties and to provide an efficient and scalable system with reasonable performance in comparison to HTTP[2]. Another challenge is to integrate a robust document revocation system that allows clients to check the validity of documents as well as to integrate the support for Web caches and content distribution networks (CDNs). This would allow us to fully exploit the advantage in comparison to HTTPS since the transport encryption of HTTPS prevents the usage of Web caches and CDNs.

[2]If a system does not aim for confidentiality, transmitted data does not need to be encrypted. So for many use-cases it would be desirable and expected that a comprehensive authenticity and integrity outperforms HTTPS since this step can be saved

2.3 Contribution

We provide with WebTrust a comprehensive authenticity and integrity framework for HTTP-based communication. WebTrust fills the feature gap between HTTP and HTTPS and provides, in particular, the following contributions:

- WebTrust leverages state-of-the art cryptographic primitives to allow users the verification the authorship of documents and to provide an end-to-end integrity proof between authors and clients. Both features are neither provided by HTTP nor by HTTPS, but they are essential in today's communication. The question whether we trust in content that we receive from the Internet, highly depends on the authorship as well as on the correctness of the transmission.
- The authorship of documents can even be proven in front of third parties (non-repudiation). There are many scenarios where such a feature is important: we could consider information that is provided by the government or a company and that changes later. Now users want to prove the original content during a trial; another scenario would be a transcript of records from our online banking account, e.g., to prove a bank transaction or our account balance to a third party.
- WebTrust integrates the so-called *Individual Verifiability*, a feature that targets at the individual verification of posts on blog-like Web sites. This provides stronger security guarantees for posts than for example the standard password-based authentication for accounts before we post information.
- Our system includes an efficient mechanism for the active revocation of documents. Documents change over time and people need to be able to verify whether a received document is still valid.
- WebTrust enables efficient data updates for already protected data.
- Content can be served very efficiently, since WebTrust supports in contrast to HTTPS both Web caches and content distribution networks (CDNs).
- WebTrust supports static, dynamic, and live-streamed content.
- Real-time streamed content can be progressively verified, i.e., on-the-fly upon retrieval. This is crucial to verify, for example, live video streams right when they are viewed.
- The WebTrust framework integrates seamlessly into the existing infrastructure and is backwards compatible with existing Web technologies. We provide a prototype implementation and conduct a performance evaluation of WebTrust to prove the feasibility of the approach.

2.4 System Overview

In the following section we provide a high-level overview of WebTrust. First, we will present our security objectives for WebTrust and describe the different settings and

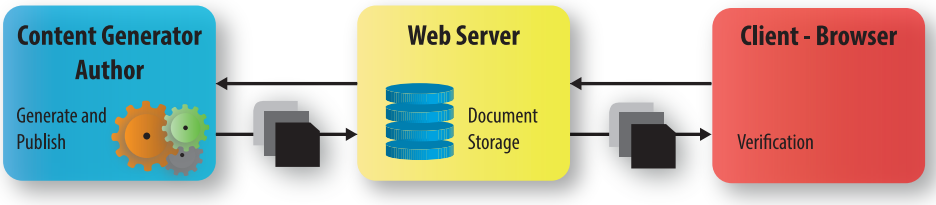


Figure 2.1: WebTrust: System Overview

content types we aim at, before we subsequently introduce the attacker model as well as the underlying assumptions of our system. The global setup of WebTrust involves three parties: a *client* integrated into a Web browser, a HTTP-based *content server* (e.g., a standard Web server with WebTrust support), and, finally, the *WebTrust Content Generator* or *author* of content (cf. Figure 2.1). The WebTrust Content Generator creates WebTrust-protected documents and uploads them to the content server. The WebTrust Client is now able to request protected documents from the Web server and to progressively verify their integrity and authenticity based on the content generator's public key during the arrival. In case the client requests dynamic content from the content server that is still to be generated by the content generator, this request is forwarded to the WebTrust Content Generator who generates the requested content on the fly. Although the content generator and the Web server do not necessarily have to be different entities, we recommend that both servers are different entities whenever possible. Having a separate content generation server reduces the risks of key exposure through content server breaches. This natural concept of splitting the content generation server from the actual content delivery server implements the well known concept of least privilege and allows us to assume a fully untrusted content server: even if the content generator trusted the content server, this still would not imply that clients do so as well.

2.4.1 Types of Content

To achieve the goal of a comprehensive authenticity and integrity framework for HTTP, WebTrust needs to support all relevant scenarios and all content types that are commonly served using the HTTP protocol:

- **Static Content:** Whenever the HTTP protocol is used to deliver *static content* that already exists at the server-side, such as a static HTML-file, a picture, or a video, the content is merely copied from the Web server to the client computer upon request. The delivered files are exactly the same for all users and among all requests.

- **Dynamic Content:** In contrast to static content, dynamic content is commonly generated on the fly and we need to differentiate between two different cases: client-side dynamic content as well as server-side dynamic content. Client-side dynamic content is generated at the client-side by embedded scripts (e.g., JavaScript) that

are used to create or fetch content. Server-side dynamic content can be treated similar as static content. Although server-side dynamic content is also generated in real-time for each single request, its delivery is just as for the static content. A typical example for server-side dynamic content is the account balance when a customer logs in to the account of an online banking Web site. The content of the same URI/resource is different for every customer as it reflects the individual account balance based on the authentication.

- **Real-time streamed content:** Real-time streamed content differs to the previously described static and dynamic content that way that its end is still unknown when the content stream is initially requested. Real-time streams are commonly used for broadcasting audio or video from live events. This scenario of real-time streamed content should not be confused with streamed content in general: the specialty is the fact that the stream is in real-time. If it is not in real-time, streaming can again be treated just like static content.

2.4.2 Security Objectives

In order to provide a comprehensive integrity framework for HTTP, several security objectives need to be fulfilled. First and foremost, WebTrust needs to provide integrity of data as well as authenticity of data with respect to the authors. In addition, WebTrust should guarantee freshness, provide a document revocation mechanism, and support non-repudiation.

1. **Integrity.** Current mechanisms used on the Internet merely provide document integrity in terms of a reliable network connection (for example, through TCP [113]). But usually when users require integrity, they want to ensure that a document was delivered without manipulation from the author to the recipient. Recent approaches have aimed at proving the integrity of documents without relying on heavyweight mechanisms such as HTTPS; however, they did not focus on the full transmission channel from authors to clients. Furthermore, they either miss important features or they do not cover all relevant use-cases (cf. our detailed discussion of related work in Section 2.11).

2. **Authenticity.** WebTrust aims at providing a proof of authenticity for Web resources with respect to their authors. This differs fundamentally from previous approaches such as HTTPS [186], which aimed at providing authenticity of connection end-points, i.e., proving the identity of the server and the client. We claim that in the majority of use-cases, especially if also the integrity of documents is of interest, users want to ensure that a particular document was delivered as intended by the author and we want to check that it stems indeed from the alleged author. In contrast, when using HTTPS-based connections we usually want to ensure that we are connected to the desired communication partner before we share sensitive information.

3. **Freshness.** Since there might be several versions of a document with a cryptographically valid signature, WebTrust requires a mechanism to ensure that users

always receive the latest document. Freshness guarantees in combination with a document revocation mechanism that a user always receives the *latest* version of a document. A man-in-the-middle should not be able to replace a document during transmission with an old, outdated, but valid WebTrust protected document.

4. **Document revocation.** A document revocation mechanism allows authors of documents to selectively revoke documents, for example when errors where found or information got deprecated. The mechanism provides users with the means to verify whether the author of a document has actively revoked the document. Since we aim with WebTrust for a comprehensive integrity framework, it needs to provide such a mechanism.

5. **Non-repudiation.** Non-repudiation allows users to prove the validity of WebTrust-protected documents to an arbitrary third party. The third party only needs to trust the provided PKI.

We do not explicitly list confidentiality as design goal of WebTrust, since it contradicts some other features desired by WebTrust, such as the support for caching. Still it might be necessary to provide data confidentiality for certain use-cases. For the transmission of data over the Internet this can easily be achieved by using WebTrust over HTTPS [186]. In case single files should be explicitly encrypted, WebTrust could simply use the encrypted files as data instead of the unencrypted. For the case of encrypted data we assume that the scenario offers already support for encrypted data such that WebTrust is only used in addition to provide, e.g., content authenticity.

2.4.3 Attacker Model

We consider for WebTrust an active adversary that is able to eavesdrop and arbitrarily modify all network traffic. However, the adversary is only able to decrypt ciphertexts or to create signatures when either the corresponding keys were created by the adversary, or when the protocol explicitly provides these keys to the adversary. We do neither assume that an adversary can efficiently brute-force or guess the decryption keys, nor that the adversary can break into systems participating in the WebTrust protocol to steal the keys. The active network adversary can selectively replace server responses by other responses with valid signatures based on his or her key. Such an adversary could, for example, be the Internet service provider that is in full control of the network connection. Furthermore, we assume that our active adversary can fully compromise the WebTrust Content Servers in our setting[3] and our assumption includes potential servers of a content distribution network (CDN). This assumption for the involved WebTrust Content Servers provides the adversary with access to all files stored on WebTrust Content Servers and allows the adversary to manipulate all requests and responses processed by the WebTrust Content Servers.

[3]This assumptions only holds true in all cases were we do not assume the Content Server and the WebTrust Content Generator to be the same entity.

2.4.4 Assumptions

In order to provide robust security guarantees for WebTrust, it is necessary to ensure that an attacker cannot gain access to the cryptographic keys used for signing content. We describe in the following the underlying requirements and assumptions of WebTrust, which need to be fulfilled to achieve our goal of providing robust security guarantees:

- The WebTrust Content Generator stores the cryptographic keys to sign content. We assume that the content generator and its underlying system cannot be compromised by an adversary.
- In case hardware is used to protect the cryptographic keys that are used for signing content (e.g., smartcards to store keys for the scenario of static content, or in case a trusted platform module (TPM) is used for dynamic content), these critical hardware components cannot be accessed or compromised by an attacker.
- In case the WebTrust Content Generator and the Content Server are the same server (this is not recommended since it implies the local storage of keys), we additionally assume that an adversary cannot compromise the Content Server. If the WebTrust Content Generator and the Content Server are not the same server, we do not state any further assumptions for the Content Server. In particular, we do not need to trust the Content Server.
- We assume a standard PKI that cannot be compromised by the adversary.
- We assume the existence of collision-resistant hash functions, the existence of chameleon hash functions, as well as the existence of cryptographic signatures. Moreover, we assume that an attacker cannot break the cryptographic properties of these schemes (for details, cf. Section 2.5).

2.5 Cryptographic Background

This section introduces the cryptographic primitives required for WebTrust. The descriptions and definitions are taken from [P7]. We start with the introduction of *Hash Functions* as well as constructions built upon hash functions and continue afterwards with the concept of *Verifiable Data Streaming* as presented by Schröder and Schröder [198]. The section closes with the introduction of *Digital Signature Schemes*.

2.5.1 Hash Functions and Constructions

WebTrust relies on two different signature providers. Both providers require collision-resistant hash functions and the one relying on the concept of *Verifiable Data Streaming* additionally requires a chameleon hash functions. Both types of hash functions including their most relevant properties as required by WebTrust will be introduced in the following. After describing collision-resistant and chameleon hash function, we will introduce *Merkle Trees* and based on the description of Merkle trees the so-called *Chameleon Authentication Trees*, which leads to the introduction of the concept of *Verifiable Data Streaming*.

The key idea behind a compressing hash function $\mathtt{h}$ is to generate a short representation (commonly referred to as the message digest, often also referred to as the fingerprint) for a large input value. We refer to the set of possible input messages with $\mathcal{M}$, to the set of possible message digests with $\mathcal{D}$, and define a keyed hash function (key k) with $\mathtt{h} := \mathtt{h}_k$ as follows [123]:

$$m \in \mathcal{M}, d \in \mathcal{D} : \mathtt{h}(m) = d.$$

An important property of hash functions is their one-wayness (also called pre-image resistance) [214]: the hash should be easy to compute $\mathtt{h}(m)$, but there should be no efficient way to compute the pre-image of $\mathtt{h}$, i.e. to compute m when given only d and the description of hash function $\mathtt{h}$.

2.5.1.1 Collision-Resistant Hash Functions

The idea of a collision-resistant hash function $\mathtt{h}_{\mathtt{cr}}$ is to ensure that for distinct input messages also the output message digests are different: the probability that an adversary can efficiently find two messages m and m' with $m \neq m'$ that lead to the same message digest d, i.e., $\mathtt{h}_{\mathtt{cr}}(m) = \mathtt{h}_{\mathtt{cr}}(m') = d$, should be negligible [214]. WebTrust requires for its design a collision-resistant hash function, but not a particular one. Any state-of-the-art collision-resistant hash function will work out. We decided to use in our implementation for the experimental evaluation for both our signature providers the SHA-1 hash function [119] and recommend for a production ready implementation to change the chosen hash function to SHA-3.

2.5.1.2 Chameleon Hash Functions

The notion of chameleon hash functions was first introduced by Krawczyk and Rabin in a technical report in 1997 [126] and officially published in 2000 [127]. In general, chameleon hash functions $\mathtt{h}_{\mathtt{ch}}$ are very similar to collision-resistant hash functions based on number theoretic assumptions. However, they have a major conceptual difference: they provide a (keyed) trapdoor to efficiently compute collisions [P7]. A chameleon hash function is typically defined by its key generation algorithm $\mathtt{keyGen}_{\mathtt{ch}}$ that is used to generate the private/public key pair $sk_{\mathtt{ch}}, pk_{\mathtt{ch}}$, the actual chameleon hash function $\mathtt{h}_{\mathtt{ch}(pk_{\mathtt{ch}})}$ that is parameterized by a public key $pk_{\mathtt{ch}}$ generated with $\mathtt{keyGen}_{\mathtt{ch}}$, and the algorithm $\mathtt{col}$ for efficiently computing a collision. So the family of chameleon hash functions can be defined as the tuple $\mathcal{H}_{\mathcal{CH}} = (\mathtt{keyGen}_{\mathtt{ch}}, \mathtt{ch}, \mathtt{col})$ over $\mathcal{M}$ and $\mathcal{D}$ [127, P7]. The chameleon hash function $\mathtt{h}_{\mathtt{ch}(pk_{\mathtt{ch}})}$ takes as input a message m as well as a randomness r and outputs the chameleon hash $d_{\mathtt{ch}}$. Its collision resistance is defined similar as for the normal collision-resistant hash function, but with the difference that the adversary gains knowledge of the public key $pk_{\mathtt{ch}}$ instead of k. New is the trapdoor functionality, which allows the owner of $sk_{\mathtt{ch}}$ to efficiently compute a collision. This can be achieved by an algorithm $\mathtt{col}$ that takes as input the secret key $sk_{\mathtt{ch}}$, a message m and its corresponding randomness r, and a message m'. The output of the collision algorithm is then r' such that $\mathtt{h}_{ch(pk_{\mathtt{ch}})}(m, r) = \mathtt{h}_{ch(pk_{\mathtt{ch}})}(m', r')$.

In general, it is possible to construct chameleon hash functions with the described properties based on a number of different theoretic assumptions [P7]. These include the

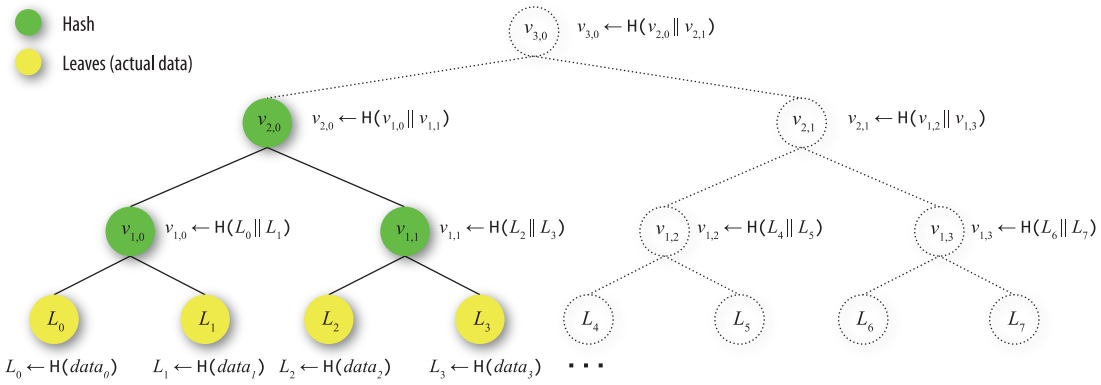


Figure 2.2: Example of a Merkle tree and its extension by one level.

factoring assumption [203], the discrete logarithm assumption [127, 18], as well as the RSA assumption [18, 105]. For WebTrust, we decided to rely on the chameleon hash scheme introduced by Ateniese and de Medeiros: its idea is based on a signature scheme introduced by Nyberg and Rueppel and this chameleon hash scheme is proven secure in the generic group model [17]. The proof assumes the hardness of a variant of the discrete logarithm problem in the cyclic group $\mathbb{Z}_p$ [17, P7]. Instead of relying on the cyclic group $\mathbb{Z}_p$, we decided to use elliptic curves [145] in WebTrust. This provides us with smaller chameleon hash values, which reduces the overhead that needs to be transmitted for WebTrust, at the cost of longer computation times on our mobile ARM-based devices[4]. For implementation details such as the library or the particular curve used by WebTrust, please refer to Section 2.7.

2.5.1.3 Merkle Trees

The concept of *Merkle Trees* [144] is commonly used to verify the correctness and the membership of single elements in larger data sets. Usually, a *Merkle Tree* is constructed as follows: all objects in a dataset that should be verified are hashed and these hashes constitute the leaves of the *Merkle Tree*. Starting with the first two hashes, now a binary tree is constructed by recursive hashes. A tree node always consists of the hash of the concatenated values of its children (cf. Figure 2.2). The tree is extended level by level until all desired leaves have been added. The root node of the tree serves now as public key for the verification of all leaf nodes inside the tree. The described binary structure of *Merkle Trees* facilitates an efficient verification since only a logarithmic amount of all tree nodes is involved for computing the proof. In order to verify a particular leaf node L against the root, it is now necessary to compute the root node based on L, which requires all adjacent nodes on the direct path from L up to the root node. Once the root node is reached, the public key should be the same as the computed value; otherwise the data object is either corrupted, or does not belong to the dataset. Figure 2.2 shows how a *Merkle Tree* is constructed.

[4]Based on results from our empirical evaluation.

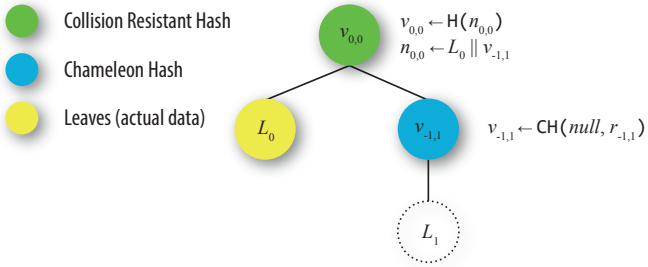


Figure 2.3: Basic example of a Chameleon Authentication Tree. The tree nodes are labeled by v and with the first index being the height (vertical position) and the second being the index (horizontal position): $v_{height,index}$.

2.5.1.4 Chameleon Authentication Trees

Chameleon Authentication Trees follow a similar idea as *Merkle Trees*: they provide a data structure that facilitates the verification of correctness for any subset of a larger dataset including a potential ordering of data. It is important to note that proving the membership of single elements or even larger subsets to the full dataset is easy, however both *Chameleon Authentication Trees* and *Merkle Trees* do not allow to efficiently prove non-membership.

The major conceptual difference between a *Chameleon Authentication Tree* (CAT) and a *Merkle Tree* is that the CAT allows to insert new leaves without the need to update the root, i.e., the public key (root node) does not change. Insertions are enabled by replacing all the hashes performed for the right nodes in a classical binary *Merkle Tree* by the previously introduced chameleon hashes (node $v_{-1,1}$) and by adding so-called dummy leafs for yet unknown leafs (L_1, cf. Figure 2.3). Since we want to later use the signed root node of the CAT to prove the authenticity and integrity of data stored in the leaves, we need to ensure that only the owner of the tree is able to change the CAT (for example, to replace dummy nodes with actual data). The properties of the chameleon hashes for computing collisions as well as the signed root node provide these security guarantees. Insertions are only feasible for the owner of the secret key for the chameleon hash functions and for the owner of the secret key for signing the root node.

A basic *Chameleon Authentication Tree* is provided in Figure 2.3. It is constructed as follows: L_0 constitutes the first leaf with actual data. The parent node $v_{0,0}$ is computed as collision-resistant hash of the two concatenated children L_0 and $v_{-1,1}$, where the right node $v_{-1,1}$ is the chameleon hash of L_1. If the data of L_1 is not yet known (Figure 2.3 shows this case), the chameleon hash is computed based on the dummy (null) leaf and the randomness $r_{-1,1}$. Later, when data is then added as L_1, a collision is computed based on the secret key $sk_{\mathtt{ch}}$, the old null leaf, the old randomness $r_{-1,1}$, and the new leaf L_1. The output of the collision calculation is the new randomness $r'_{-1,1}$ that leads together with L_1 to the same chameleon hash for $v_{-1,1}$ as the null leaf with $r_{-1,1}$ did. This ensures that the root node of the chameleon hash tree is not updated.

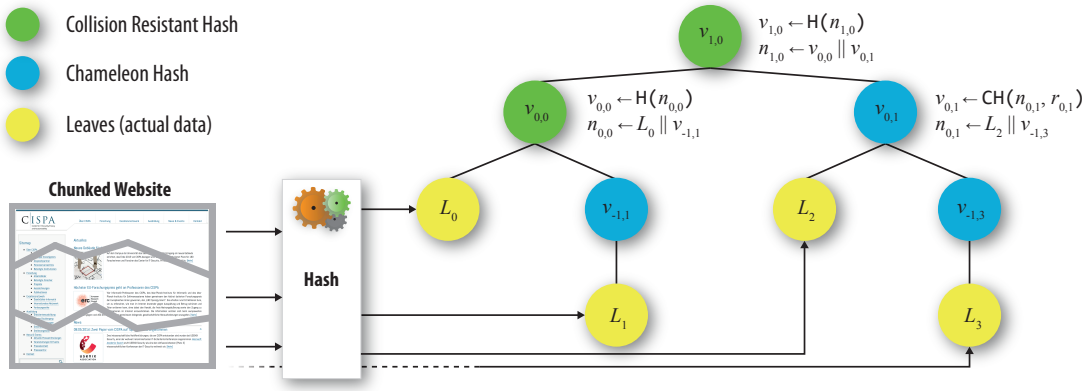


Figure 2.4: Typical usage of Chameleon Authentication Trees in WebTrust. The example contains four data leafs $(L_0, \ldots, L_3)$ (P7).

2.5.2 Verifiable Data Streaming

The *Verifiable Data Streaming* (VDS) protocol is based on the previously introduced *Chameleon Authentication Trees* and was introduced by Schröder and Schröder in 2012. Originally, the verifiable data streaming protocol was designed to stream data from a computationally weak client to a powerful server. The core idea of WebTrust is not the outsourcing of computations as for VDS, but we can nicely adapt the VDS protocol for our setting.

2.5.2.1 Adaptation of VDS for WebTrust

In the WebTrust protocol, the content generator takes the role of the computationally weak client in the original VDS protocol. The server from the VDS protocol is in our scenario the untrusted Web server, which serves the data to the public Internet. Clients can now publicly verify data that they receive from the Web server. Figure 2.4 shows how the *Chameleon Authentication Tree* for the VDS protocol is constructed. The file we want to protect (for example, a Web site) is split up into segments and each segment constitutes a leaf (L_0 to L_3 in Figure 2.4). The left leaves are always a hash of the concatenated children (green, uses a collision-resistant hash function), the right leaves are the chameleon hash of the concatenated children (blue). Since the CAT is always extended to the right side, the root node is also computed by a collision-resistant hash function. It either serves directly as the public key for the verification of the WebTrust protected file, or it is part of the public key. For our scenario, the root node of a per-file CAT is the leaf of another CAT whose root node serves as the public key for the full dataset. In case the full file that should be added to the CAT is not yet known (e.g., for a video live stream), a CAT for the expected final file size is created and dummy leafs are added as placeholders. The dummy leafs consist of chameleon hashes that do not yet contain real data, but dummy values (cf. Figure 2.5). Once the final data is known, a collision is computed as previously described and the actual data is added. Replacing these dummy leafs or adding the data does not involve an update of the root

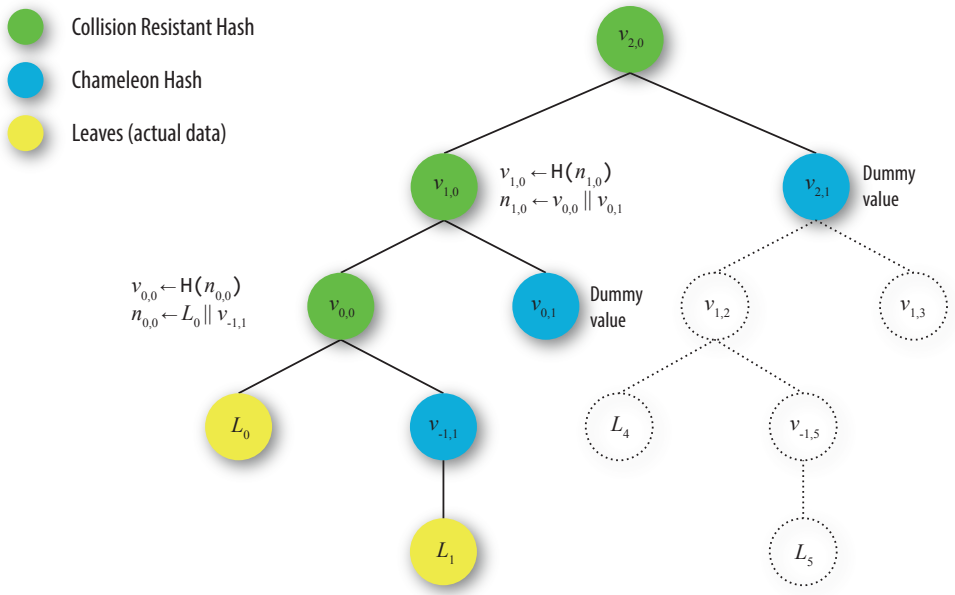


Figure 2.5: Example for appending new data to existing chameleon hash nodes in a chameleon authentication tree.

node and, thereby, of the public key. However, if the initial CAT was created to small, it can be dynamically extended. But this involves an update of the public key since a full layer is added on top of the root node.

2.5.3 Digital Signature Schemes

Finally, WebTrust requires digital signature schemes. A digital signature scheme allows to compute a signature sig for a message m based on a secret key sk. Afterwards, every user with access to the corresponding public key pk is able to verify the correctness of the signature sig. WebTrust relies on the RSA signature scheme [188], since it provides a very fast verification routine, which is in particular on resource-constrained mobile devices a huge advantage. The RSA signature scheme is secure against the standard notion of existential forgery under chosen message attacks [123, P7]. It uses as underlying mathematical structure composite order groups [P7] and the security of the RSA scheme can be proven in the random oracle model [28, P7].

2.6 System Details

In the following section we describe in detail how WebTrust achieves its design goals of a comprehensive authenticity and integrity framework for HTTP. We discuss in detail how the system is set up and how the cryptographic primitives introduced in Section 2.5 are used to achieve each of the desired features for WebTrust.

2.6.1 System Setup

The system setup of WebTrust consists of client-side and server-side components and involves three different entities: the author of content which we refer to as the *Content Generator*, the *content server* responsible for actually delivering content, and finally the *WebTrust Client* who requests and verifies WebTrust protected content. Depending on the usage scenario, the requirements of the content generator, and the content type that should be served, WebTrust leverages a different system setup to always achieve the maximum amount of security possible under the given conditions.

All setup scenarios have in common that data to be served is split into smaller segments and that WebTrust signatures are sent in an interleaved order with these segments. We will now first introduce the different setup options of WebTrust before we discuss our two different signature providers (namely *VDSECC* and *RSA-Chaining*, they are supported by all our setups) and the individual WebTrust features.

2.6.1.1 Static Content

The preferred and most secure setup is the one for static content where the content generator and the content server are different entities (cf. Figure 2.6). This setup allows a very effective protection of the keys used by our signature providers, since the secret key sk_{static} for signing content is only required when either new documents are created, or existing documents are modified. Figure 2.6 shows how a file containing a Web site is split into the segments s_1, s_2, and s_3. Our signature provider (described as `Sign`) processes the individual segments and interleaves the WebTrust signatures for publishing each segment with the corresponding signature subsequently to the *content server*. Clients are now able to request documents from the content server and to verify the incoming segments one by one. Using a secure environment for storing the secret key sk_{static} and for signing documents is of course only a recommendation. The decision to use such an environment will most likely depend on the importance of the content served. WebTrust would also work if both signing and key storage are done in the untrusted environment of a normal operating system with the obvious impact on the security guarantees.

2.6.1.2 Dynamic Content

In contrast to the setup for static content, the setup for dynamic content requires the signing key to always be present. Dynamic content is usually either generated individually each time it is requested, or at least individually per user. Therefore, signatures cannot be precomputed upfront and need to be created right after the content has been generated and before it is delivered. Achieving a meaningful protection for the signing key $sk_{dynamic}$ is, therefore, a central challenge for serving dynamic content with WebTrust protection. In general, one could think of two different ways to set up a WebTrust system that protects dynamic content.

The first possible setup is shown in Figure 2.7. The WebTrust Content Generator can be considered as a modern database back-end for a Web server, but with the major difference that the full dynamic site is already created at this back-end (for example, if

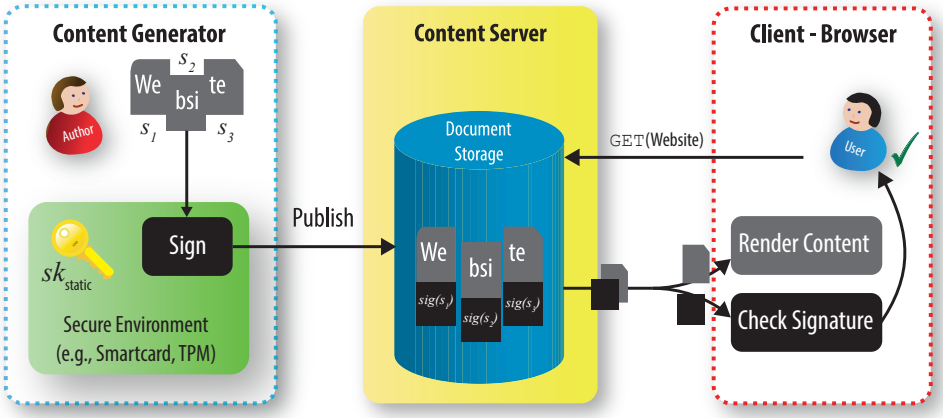


Figure 2.6: WebTrust: System setup for static and streamed content

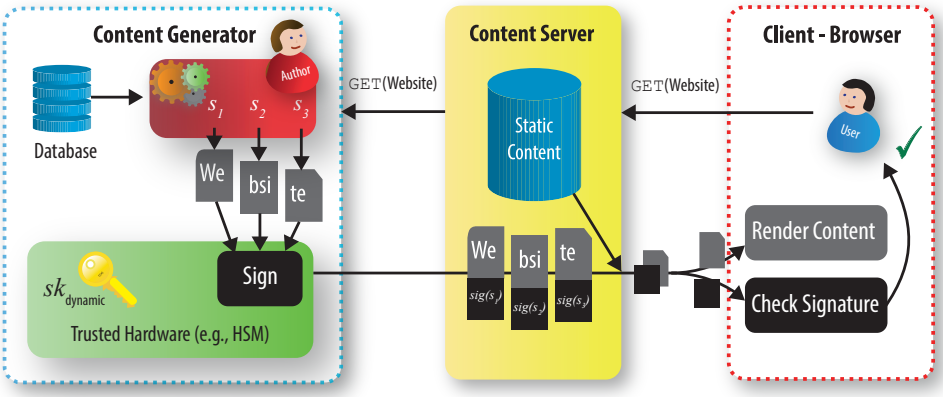


Figure 2.7: System setup for dynamic and real-time streamed content in the scenario where the WebTrust Content Generator ≠ Content Server

dynamic content is loaded in an individual `iFrame`). This way, we can already add the WebTrust protection at the WebTrust Content Generator. In our example presented in Figure 2.7, the content generator takes input from the database and creates a Web site, which is, as previously for static content, split into the segments s_1, s_2, and s_3. They are processed in a secure environment, such as a hardware security module that also stores the secret key used for signing content $sk_{dynamic}$, and subsequently delivered with the individual signatures to the Content Server. Upon arrival, the Content Server forwards the incoming segments to the client that initially requested the Web site. Static content loaded by other `iFrames` would be stored persistently on the content server. Security-wise, this is the preferred setting since the WebTrust Content Generator and, thereby, also the secret key for signing content $sk_{dynamic}$, can be protected more efficiently against adversaries than the content server. However, as shown by Moyer et al. [149], this setup will also introduce some additional overhead. Nevertheless, this setup is the one used

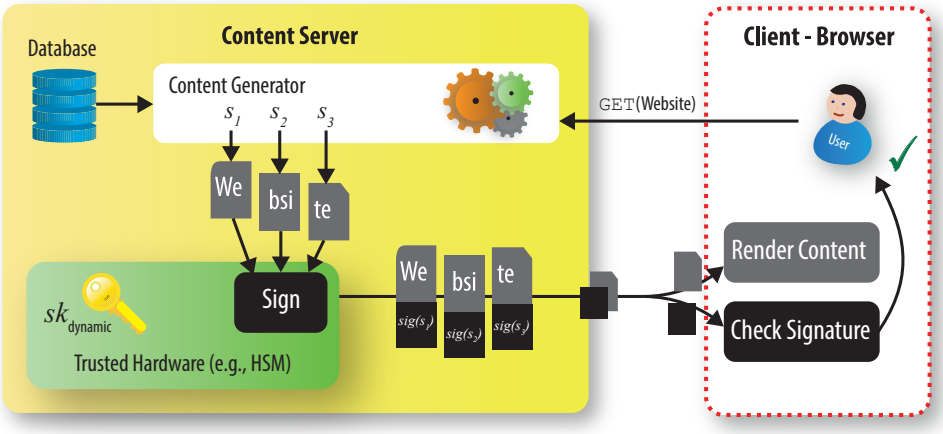


Figure 2.8: System setup for dynamic and real-time streamed content in the scenario where the WebTrust Content Generator = Content Server

by many big Web sites that serve dynamic content and we recommend to further use for the WebTrust Content Generator trusted hardware to reduce the risks of both a server breach and the exposure of the signing key. Using this setup allows us also to refrain from making any particular trust assumptions on the content server.

The second possible setup is shown in Figure 2.8. This time the WebTrust Content Generator and the content server are considered as the same entity: they are running on the same server. All resources used (e.g., database entries, the secret key used for signing content $sk_{dynamic}$, already signed static Websites, etc.) reside on this server. Moreover, all required processing of content is also done on this server. If this setup with a single server for both content processing and content delivery is chosen, we highly recommend the usage of trusted hardware to protect the secret key used for signing content $sk_{dynamic}$ to reduce the impact of a server breach (cf. Section 2.6.4 for details on hardware security modules). Additionally, the setup has impact on our trust assumptions. We can no longer consider this server untrusted. We do not recommend this setup; nevertheless, this is still a popular setup for small Web sites serving dynamic content. Besides size and costs, another reason for such a setup could be time-critical applications that serve dynamic content.

The granularity at which dynamic content is signed is similar as for the static content since it is still further split up into individual segments. However, supporting dynamic content with WebTrust introduces besides the protection of the secret key for signing content ($sk_{dynamic}$) another challenge: Our WebTrust signature providers need to support an on-the-fly signing of individual content without yet knowing the end of a full file. Ensuring integrity, checks on the order and freshness, and still providing the possibility of a progressive client-side verification are not features that can be easily achieved.

2.6.1.3 Streamed Content

Streamed content is content that is usually consumed immediately upon retrieval. Clients do not wait until the full file has arrived before the content is rendered. Additionally, one also does not have to know the beginning of the requested content. One could simply tune in at an arbitrary position. For WebTrust, streamed content falls into two categories: First of all, streamed content can be an already existing static file that is delivered to a client (for example, a large video file). Due to the construction of our signature providers, which allow the verification of individual segments of a large file at any time without knowing predecessor or successor segments, we can treat such streamed static files just like normal static content (cf. Figure 2.6). Second, we consider streamed content that is provided in real time such as a live video stream from an important incident. In contrast to the previous form of streamed content, the end of the file is not known when first segments are already sent to a client. In particular, these segments should be verified at the client-side before the content generator knows the end of the file. Again the construction of our signature providers allows us to handle such content very efficiently. It is treated just like dynamic content. In order to facilitate the WebTrust usage by authors of dynamic content, we imagine for the future that the infrastructure for signing content is available to recording devices: WebTrust could, for example, be integrated into professional cameras as they are used to report from important incidents. Once the user successfully passed the authentication at such a camera, real-time streams can be protected directly from where they are generated.

2.6.2 Signature Providers

Our signature providers are central to both the utility and the security guarantees of WebTrust. Their design allows us to fulfill the requirements of a comprehensive authenticity and integrity framework for HTTP. Currently, WebTrust supports two different signature providers, namely *VDSECC* and *RSA-Chaining*. However, the system can be extended by new signature providers as long as our requirements are met and the defined security goals a reached.

2.6.2.1 VDSECC

VDSECC leverages the concept of *Chameleon Authentication Trees* as they were introduced in Section 2.5 to achieve our first signature provider. Content (let us assume for the following an `index.html` file as example) is first split into segments $s_1, s_2, \ldots$ and subsequently hashed. Once the segments of the `index.html` file haven been hashed, they are added as leafs $L_1, L_2, \ldots$ to the *Chameleon Authentication Tree* (cf. Figure 2.4). The L_0 leaf contains meta information such as the URI of the `index.html` file, its creation date, as well as an expiration date or time to live. The meta information does not need to include an order of the subsequent segments, since the ordering is implicitly ensured by the *Chameleon Authentication Tree*. All inner left nodes of the chameleon authentication tree are now constructed by using a collision-resistant hash function and all inner right nodes are computed by using a chameleon hash function (cf. Section 2.5 for the cryptographic background). Finally, either the root node of the *Chameleon*

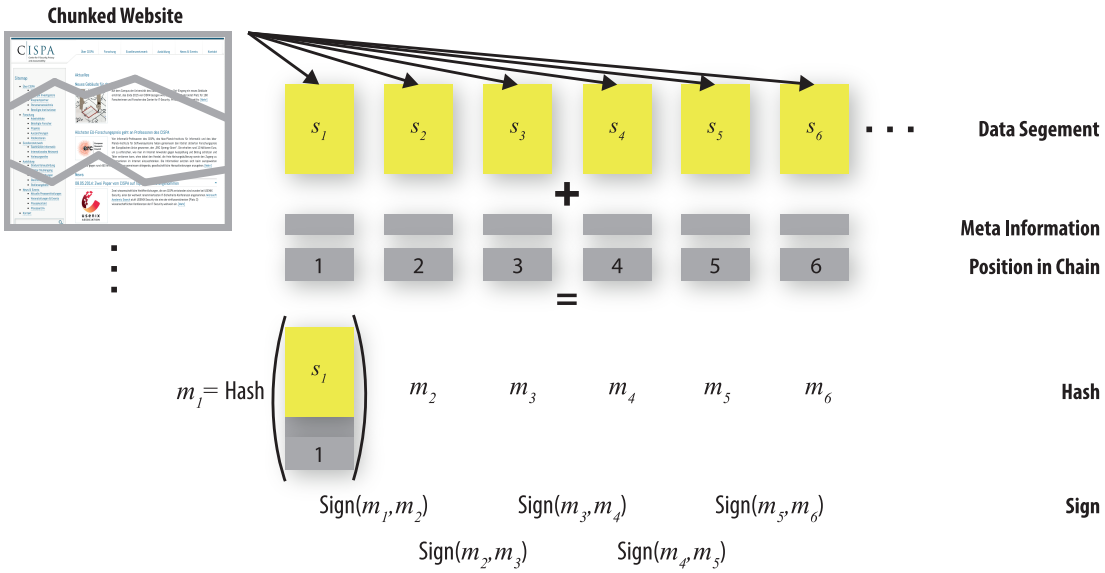


Figure 2.9: WebTrust: Signature Provider RSA-Chaining

Authentication Tree is signed with the author's secret key, or it is added to the additional *Chameleon Authentication Tree* required for content revocation (from now on referred to as the *Authentication CAT*, for details cf. Section 2.6.3.3). The client-side verification of *VDSECC* is based on the reconstruction of the CAT's root node as well as on the verification of the root node's signature.

2.6.2.2 RSA-Chaining

The second signature provider of WebTrust leverages the RSA signature algorithm [188] and uses a mixture of a hash chain and the concept of a doubly linked list. We provide an overview of this *RSA-Chaining* signature provider in Figure 2.9. In case of *RSA-Chaining*, content is again split into single segments $s_1, s_2, \ldots$ and the same meta information as for the *VDSECC* signature provider is added to *every* single segment: the URI of the resource, the creation date of the resource, as well as an expiration date or time to live. In addition to this meta information, we add for *RSA-Chaining* also the position of the individual segments. Afterwards, we gain m_i by hashing each segment together with its meta information and its position in the chain by a collision-resistant hash function. These hashed segments are finally used to create the signatures based on *RSA-Chaining* by signing always two consecutive segments: $Sign(m_i, m_i + 1)$. In order to also support content revocations by *RSA-Chaining*, we further introduce special WebTrust content revocation lists, so-called *WT-CRLs*, to which we add revoked segments. The client-side verification of signatures generated by this signature provider is based on the standard signature verification of RSA. Overall, this construction of *RSA-Chaining* allows us to support all features required by WebTrust.

2.6.3 Features

WebTrust was designed to enable (1) the verification of both *authorship and integrity* for arbitrary documents sent via HTTP with respect to the authors. It even allows (2) proving both the authorship and the integrity of documents in front of third parties (*non-repudiation*). Clients can (3) perform an on-the-fly verification of incoming packets (*progressive verification*), and WebTrust allows content generators to (4) *actively revoke* documents as well as to provide freshness information. In order to reduce the load at the content server and to distribute the network load, WebTrust was further designed to (5) support Web caches and content distribution networks (CDNs). Finally, WebTrust provides (6) a mechanism for dynamic content updates and enables (7) the individual verifiability to provide guarantees for individual posts on blog-like Web sites. In the following we explain for all features how they are actually achieved.

2.6.3.1 Authorship and Integrity

The WebTrust protection for documents is based on the two WebTrust signature providers *VDSECC* and *RSA-Chaining*. In case of *VDSECC*, the author (content generator) signs the root node of the chameleon authentication tree with his secret key. For *RSA-Chaining*, the hashes of two consecutive data segments are signed using the secret key of the author. So to verify the authorship of a document, clients need to be able to reliably link the public key for verifying the created signatures to the author identity. The certification authority certifies the identity of the author by checking the real world identity of the author against the information provided in the certificate of the author upfront. The author certificate is only signed if the stated information is correct. Based on the PKI and the signature providers, we can therefore enable the verification of the authorship. The signature also ensures the integrity of data, since a manipulation would lead to a failing verification.

2.6.3.2 Non-Repudiation

Using signatures based on keys that are part of a public key infrastructure provides also an efficient mechanism for achieving non-repudiation. Central to all WebTrust content protections are the two signature providers *VDSECC* and *RSA-Chaining*, which both allow the verification of authorship and integrity if clients are in possession of the required public keys. The only additional requirement is that clients have trust in the linkage between the public key for the verification and the identity stated for the key. The verification of documents is in general also possible for all third parties: Every WebTrust protected content block is combined with all the information required to perform the verification. If a third party receives both content and the corresponding protection, the third party only needs to trust the PKI that provides a certified key of the content author to perform a reliable verification of the content.

2.6.3.3 Document Revocation and Freshness

WebTrust relies for the signature creation on two conceptually different signature providers and both have their individual document revocation mechanisms. Providing

such a mechanism is also a crucial prerequisite for achieving efficient secure content updates.

VDSECC uses a second *Chameleon Authentication Tree*, the *Revocation CAT*, which functions as a content revocation tree for all files that are signed by the same secret key of the same content author. We provide an overview of the construction of this *Revocation CAT*) in Figure 2.10. The root node of each individual file's CAT is added as a leaf to this revocation tree. In the example in Figure 2.10, the root node of the *Chameleon Authentication Tree* for file one ($v'_{0,0}$) is added as leaf L_1 to the *Revocation CAT* and the root node of the *Chameleon Authentication Tree* for file two ($v''_{0,0}$) is added accordingly as L_2. If, for example, a segment of file one is now updated, the root node of the CAT for file one changes as well, and so does the root node of the *Revocation CAT*. In both CATs (the CAT for file one and the *Revocation CAT*), such an update of a segment leads to updating a logarithmic amount of nodes in both trees [P7]. In case a file should be intentionally revoked without providing a new file, we can simply replace the root node of the file to be revoked in the leaf of the *Revocation CAT* by a revocation notice, which also ensures a update of the root node of the *Revocation CAT*. The updated root node of the *Revocation CAT* is signed again in both cases. The big advantage of this particular revocation mechanism is the fact that the revocation information does not grow linearly with the amount of revoked segments, but stays constant.

In contrast to *VDSECC*, *RSA-Chaining* does not use the *Revocation CAT* but relies on special WebTrust content revocation lists (*WT-CRLs*). Every time a segment is updated or revoked, the hash of the old or to-be-revoked segment is added to these *WT-CRL*. So in contrast to the space efficient *VDSECC* solution, the *WT-CRL* grows linearly with respect to the amount of segments that need to be revoked. Intuitively, one might consider using *RSA-Chaining* as signature provider in combination with the *VDSECC* revocation mechanism. However, this would result in a very large CAT: It would be basically a CAT that includes the hash of all segments of all files in one single CAT, which could already serve as protection for itself. To this end, the intuition of a hybrid approach does not provide a recommended solution.

Freshness is provided both for the *VDSECC* signature provider and the *RSA-Chaining* signature provider by adding a time to live or an explicit expiration date to the meta information. In case of *VDSECC*, this is provided in L_0, for *RSA-Chaining* this is added to every single segment.

2.6.3.4 Progressive Verification of Content

In order to enable a progressive (on-the-fly) verification of content we need a mechanism that allows the verification of authorship and integrity for every individual block. WebTrust achieves this by splitting up content into individual segments and providing each block with an individual protection: the signatures for individual segments of both our signature providers (*VDSECC* and *RSA-Chaining*) can be checked immediately after arrival. While for *RSA-Chaining* this check is a standard RSA signature verification against the public key of the author, the situation is slightly more complex for *VDSECC*: The verification of individual segments requires to reconstruct the root node of the

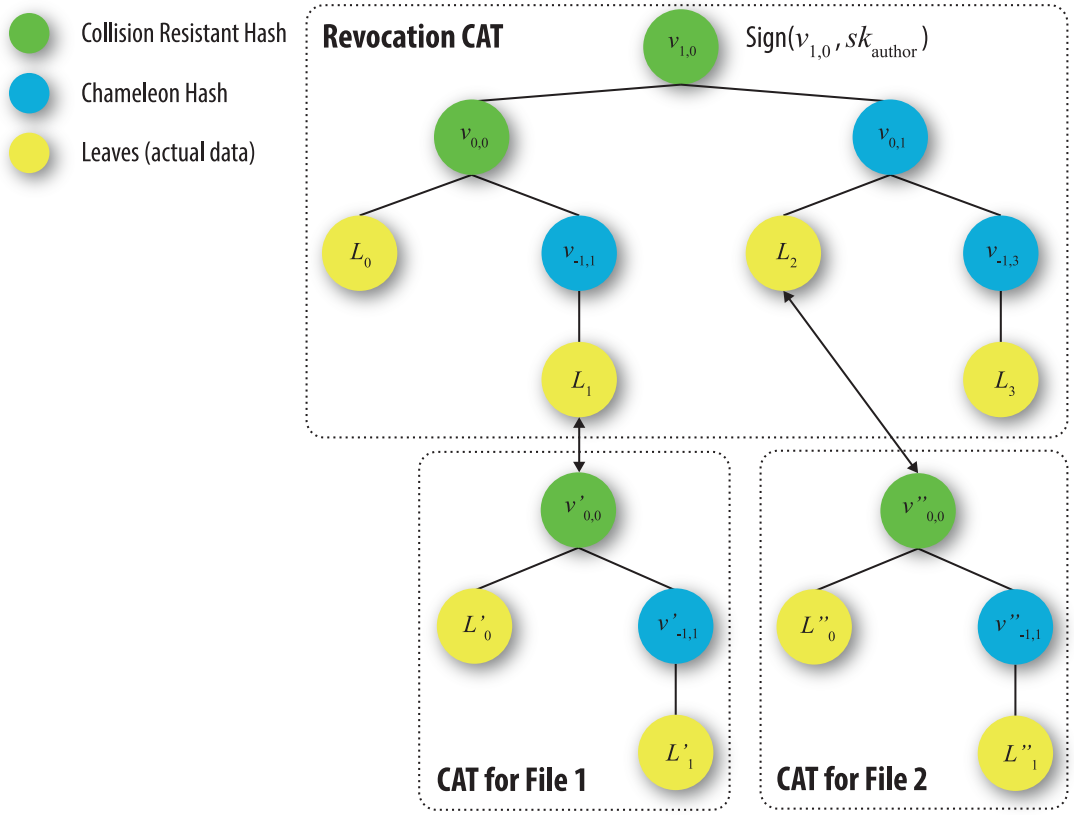


Figure 2.10: WebTrust: Revocation Tree for Signature Provider VDSECC

Chameleon Authentication Tree for the file, and potentially also the reconstruction of the previously mentioned *Chameleon Authentication Tree* for file protection and revocation. The reconstruction of these two *Chameleon Authentication Trees* requires all neighboring nodes on the path to the respective root nodes to compute the missing hashes. Both *VDSECC* and *RSA-Chaining* allow to sign and send any block i without yet knowing the subsequent block $i+1$. In general, WebTrust supports the progressive verification of all content types (static, dynamic, and real-time streamed content) and it even allows to verify any block i of a file without needing to receive the segments $i-1$ or $i+1$ before.

2.6.3.5 Caching and CDN-Support

Central for the support of Web caches or content distribution networks (CDNs) is that the WebTrust protection of a file is equal for every user and that content is always split into segments of equal size for every request. This is the same prerequisite as for a local signature caching at the content generator for dynamically generated content. Since WebTrust-protected content as well as its signatures are usually transmitted as cleartext messages, content and signatures are the same for all registered users.

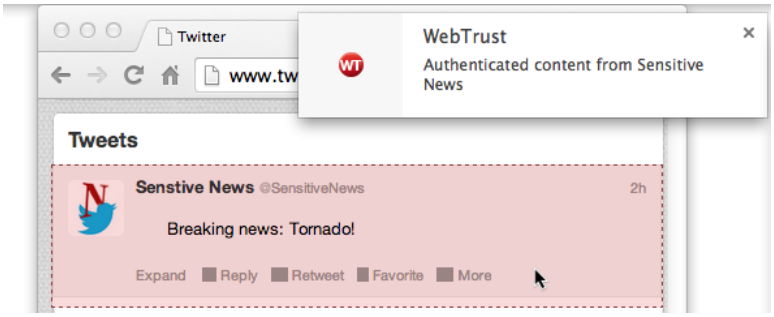


Figure 2.11: WebTrust Extension for Chrome: Informing the user about the verified authorship of an embedded tweet four a custom Twitter-like Web site (P7)

2.6.3.6 Content Updates

Both our signature providers allow efficient content updates. As described in Section 2.5, VDSECC is based on *Chameleon Authentication Trees*, which allow an efficient update of data leafs since only a logarithmic amount of the total tree nodes needs to be updated. The update process of a data leaf involves an update of the root node, which either requires to update the signature of the root node or to propagate the update through the previously introduced *Revocation CAT*. For *RSA-Chaining* it is possible to replace single segments in the chain by simply re-computing two signatures: the first signature that needs to be re-computed signs the hash over the predecessor of the new block and the new block itself, the second signature that needs to be re-computed is the one for the hash of the new block and its successor. In both cases it is not necessary to re-compute the full data structure of the protection, so we do neither have to fully reconstruct the *Chameleon Authentication Tree* nor the RSA chain.

2.6.3.7 Individual Verifiability

A central motivation for WebTrust was to provide users with the possibility to verify the authorship of content. Of course, it should also be possible to combine content of different authors in one page and to verify each part individually. WebTrust provides such an *individual verifiability*: each author's content needs to be loaded into an individual IFrame. A Web site, such as a Twitter-like page may contain now several IFrames, each with content from a different author. Whenever an IFrame is supposed to be loaded, the browser triggers a separate WebTrust-protected HTTP request. The user is finally able to receive visual feedback about the verification result of each IFrame by hovering the corresponding area as shown in Figure 2.11.

2.6.4 Key Security

WebTrust makes use of custom signature providers that allow content generators to generate WebTrust-protected content. Both our signature providers, *VDSECC* and *RSA-Chaining*, rely on classic signature schemes such as RSA [188]. Besides the security guarantees provided by the cryptographic properties of these schemes, such signature schemes also crucially rely on a robust protection of the secret key. If an attacker got

access to such a key, one could simply create new signatures for modified content. A very high level of protection for such keys is commonly achieved if these keys are stored in so-called Hardware Secure Modules (HSMs, cf. Figure 2.8 for our WebTrust setup serving dynamic content: the overview integrates an HSM), which are inaccessible for most kinds of attackers. The best protection for our setup can be achieved for static content, since the key does not have to be always present at the server. We recommend in this case to store the key in a smartcard and to only plug it in when content should be signed. The weakest security guarantees for WebTrust can be provided for dynamic content if the content server and the WebTrust Content Generator are the entities: In this case the key needs to be stored in the HSM on the delivering content server. A server breach would now have the impact that an attacker could use the HSM with the key stored as a signing oracle. However, the attacker could not access or copy the key directly. This basically achieves the same level of security as on can achieve if the Web server stores the private server keys for the HTTPS/TLS authentication, which is common practice on the Internet and without alternative. In order to still protect the WebTrust Content Generator, one could still follow the approach by Moyer et al. presented for the Spork system [148, 149] and use runtime attestation to try to detect a server breach.

2.7 Implementation

In the following, we introduce the implementation of WebTrust at the server-side and at the client-side. Both the server-side and the client-side implementation are still to be considered as prototypical implementations that are supposed to provide insights on the performance of our scheme, but yet are missing some of the functionality. In particular, our content generator as well as the Web server are implemented as one single machine and we have not yet implemented the document revocation mechanism as well as the extended PKI as introduced in Section 2.6.

2.7.1 WebTrust Server

Our server-side implementation of WebTrust consists of a patch to the JAVA-based Apache Tomcat Web server 7.0.39 [11]. The WebTrust patch to the Tomcat server extends the existing handling routines for the HTTP 1.1 chunked transfer encoding [73], which includes the routines responsible for handling incoming requests as well as the routines for preparing and sending the response message. Most of our modifications target the chunked output filter (`ChunkedOutputFilter.java`) of Tomcat where we integrate our signature creation routines. The actual WebTrust protection is embedded as so-called chunk extensions, which are defined in RFC 7230 for the HTTP 1.1 chunked mode. The general structure of chunks looks as follows (the definitions are based on the RFC 7230 [73]):

```
chunk-size [ chunk-ext ] CRLF
chunk-body
```

The chunk starts with the size of the full chunk in hexadecimal form and it is either followed by chunk extensions of the form *;ext-name=ext-value;ext-name=ext-value;...*, or the line is ended by a carriage return and line feed. The chunk header is then followed by the actual data in the chunk body. Chunk extensions may omit the `ext-value` and their number is not limited. Our chunk extension for WebTrust currently starts with `SIG=` and it is followed by our WebTrust signature in Base64 encoding [120]. A typical example of a chunk header including the WebTrust extension now looks as follows [P7]:

```
AC9;SIG=8CD3ABU8ULS2KMDN4HW3NK6A5BPP84HB6A7CC
```

The `AC9` value at the beginning is the hexadecimal representation of 2761 bytes and defines the length of the full chunk. Leveraging the chunked mode of HTTP 1.1 and embedding all data required for a later verification as standard conform chunk extensions, provides us with full backwards compatibility and allows for a seamless integration. We verified that extensions to the HTTP 1.1 chunked mode are in general supported by all major Web browsers such as Apple Safari, Google Chrome, Microsoft Internet Explorer, and Mozilla Firefox, as well as by popular tools used for HTTP requests such as cURL, GNU Wget, or Java [P7]. Clients that do not yet support WebTrust will simply ignore our chunk extensions and will process and render the HTTP response as usual. In order to notify clients of the WebTrust protection in a particular HTTP response, we introduce two additional headers:

```
Content-Verification-Scheme: 1.0/SHA1-VDSECC
Content-Verification-Key: 61KJHQ1J4NED97NBP2SJ44FP0
```

The first header informs clients that the incoming response is protected with WebTrust version 1.0 and based on `SHA1-VDSECC`, i.e. VDS based on elliptic curves and the hash function SHA1. The version information is integrated for protocol updates in the future and does not have any impact at the moment. Besides `SHA1-VDSECC`, the server of course could have also used the other signature provider `SHA1-RSA`. The second header informs the client about the public key required for the verification.

In order to improve the performance at the server side, our implementation includes several optimizations. First of all, we introduced a signature cache at the content generator to prevent expensive re-computations of signatures. For static content, the signature cache was implemented based on the default servlet of Tomcat, which is commonly used for directory listings and static HTML-files [11]. Using the default servlet ensures that every response to the same URI request leads to the same chunks with equal size, which is the crucial basis to achieve an efficient caching. When looking at dynamic content, the situation becomes slightly more complex. The involved servlets decide by themselves when data is actually sent to the network. So whenever these servlets call for example `flush()`, data is not sent to the network but to the WebTrust output filter where data is first accumulated up to a specified size and signed by our WebTrust signature providers before it is actually sent to the network [P7]. The accumulation of bytes in this WebTrust output filter enables caching also for dynamic content: it simply ensures that whenever two separate runs of the same function on the same input are executed and lead to the same output, this also leads to chunks of the same content and size. And this fulfills again the requirements for an efficient caching.

Another performance optimization at the server side ensures that WebTrust is really only used if the client requested its usage: we only activate the WebTrust routines for the response message if the client request included the corresponding header for using WebTrust as follows:

```
Accept-Content-Verification: SHA1-VDSECC
```

The currently supported signature providers for WebTrust are `SHA1-RSA` and `SHA1-VDSECC`, so instead of `SHA1-VDSECC` the client could also request the usage of `SHA1-RSA`. In case the `Accept-Content-Verification` header is set, the server prepares a response with the according protection and sends it to the client.

In order to implement our signature providers `SHA1-VDSECC` and `SHA1-RSA` at the server-side, we leverage Oracle's standard Java cryptography providers [163], namely the `Sun` and the `SunRsaSign` cryptography provider, for all non-elliptic curve primitives. All functionality that involves elliptic curve operations for VDSECC is based on the Bouncycastle cryptography provider [32] and the implementation for VDSECC is based on the Java library for chameleon authentication trees as introduced in [P7].

2.7.2 WebTrust Client

The core functionality of our WebTrust client is implemented as a patch for the open-source variant of the Chromium browser version 29 [217], the open-source project that also forms the basis for the popular Google Chrome browser. The patch is complemented by a chromium extension for prototyping the individual verifiability feature of WebTrust.

The patch to Chromium extends the handling routines for the HTTP 1.1 chunked mode in Chromium for parsing and verifying the WebTrust protection of incoming packets. Besides, the routines for issuing a HTTP request are modified and allow now to send a request with our new WebTrust header `Accept-Content-Verification` including the preferred protection algorithm. This header informs the Web server of the desired WebTrust based protection for the corresponding response and initiates, if supported by the server, the WebTrust routines. Besides the modifications to the processing routines of Chromium, we additionally modified the UI of the Chromium browser: It includes now a visual indicator for the user whether the authenticity and integrity verification of the incoming HTTP response message was successful. Regarding the authenticity, this visual indicator will also indicate in the future whether the key used for singing the data is a valid and trusted key in the PKI. Nevertheless, users will still have to check manually, whether the author listed in the certificate is the expected author.

Our browser extension for WebTrust provides a mockup implementation of the individual verifiability as discussed in Section 2.6.3.7. The individual verifiability works on the granularity of individual HTTP requests for `IFrames`. Figure 2.11 on page 25 shows how the verification results of individual `IFrames` are presented to the user.

Similar to the server-side implementation, also the WebTrust client makes use of external libraries for the cryptographic primitives. In particular, WebTrust relies on the OpenSSL library [162] and uses the C++ library for *Chameleon Authentication Trees* as introduced in [P7].

2.8 Security Evaluation

The goal of WebTrust is to achieve a comprehensive authenticity and integrity framework for HTTP with robust security guarantees. In the following, we discuss the individual design goals and supported features of WebTrust with respect to their security guarantees.

2.8.1 Integrity, Authenticity, and Non-Repudiation

All three properties, the integrity protection of content, the provable authorship of content, as well as the possibility to proof this correctness and authorship in front of third-parties, rely for their security guarantees on the cryptographic properties of the primitives used.

Both our signature providers ensure the integrity of content by hashing individual segments with a collision-resistant hash function and by including these hashes into the signatures. The correctness of this integrity protection primarily relies on the collision resistance property of the hash function used. An attacker should not be able to efficiently compute a second data segment that results in the hash of the protected data segment. Otherwise, the attacker could simply replace the data segment by the computed segment without anyone being able to recognize. Since we assume that the attacker cannot break the collision resistance property, the only possibility left would be to replace both the data segment and the hash. WebTrust includes the hashes of segments into the signatures to ensure that an attacker cannot simply replace the segment including its hash, so that such a replacement would also require the attacker to forge the corresponding signature. But the attacker has neither access to the secret key of the original content author to create such a signature, nor is the attacker able to break the cryptographic signatures. Therefore, it is impossible to forge a valid signature for the hash of the data segment to be manipulated, which also renders it impossible to compromise the correctness of the integrity proof.

The proof of authorship for WebTrust is based on the same signature that is used to protect the hashes for proving integrity. The signature can be verified against the public key of the content generator, which is certified by a PKI. Under the assumptions that the signature algorithm itself cannot be broken, the secret key is inaccessible for the attacker, and the correctness of the certification by the PKI cannot be compromised, it is impossible for an active attacker to generate content in the name of the content generator. Therefore, an attacker is not able to replace WebTrust-protected content with own, manipulated packets that are validly signed by the content generator. Replacing content would either lead to a broken signature, or, if both content and signature are replaced, to a signature that verifies against the public key of the attacker: the only chance for an attacker to create valid signatures is to use his own key for signing new or manipulated content. If an attacker tries to replace protected segments by other segments that are validly signed by the content generator but stem from a different URI, clients will recognize this content replacement since the URI is part of the protected meta data. In case different content is served via HTTP based on session information

from the same URI [5], manipulations can also be recognized: the session information is part of the HTTP header, which is also protected by WebTrust.

In case an attacker tries to replace protected segments by older versions of the same segment or document, this can be recognized based on two mechanisms: our freshness information and the content revocation mechanism allow clients to verify that a document is still valid. Both our freshness and our revocation mechanism will be explained in the following section.

Our support of *non-repudiation* is given by the construction of our signature providers *VDSECC* and *RSA-Chaining*. Their security guarantees are both based on cryptographic signatures that are verified against the public key of the content generator. Since third parties are able to verify the certification of the public key required for the signature verification of WebTrust-protected content, we can easily guarantee the correctness of WebTrust protected content as well as the authorship of this content to third parties. The security guarantees of the *non-repudiation* are based on the correctness of the integrity proof and the PKI, as well as on the provable authorship based on signatures.

2.8.2 Freshness and Content Revocation

WebTrust incorporates freshness information into its signatures by adding a time-to-live or an expiration date into the meta information. This prevents an attacker from replacing content segments in a transmission by an older but correctly signed version of the segment. For the *VDSECC* signature provider, the meta data with the freshness information is provided in the first leaf (L_0), whereas for the *RSA-Chaining* signature provider the meta information is added to each single segment. Both signature providers protect the included meta information based on signatures such that an attacker cannot manipulate or replace this information.

In case a document is supposed to become invalid before the expiration date is reached or the time-to-live has expired, content generators can use an active content revocation mechanism to invalidate documents. The *VDSECC* signature provider integrates an implicit content revocation mechanism, either based on the individual CAT of a document, or based on the *Revocation CAT*. Whenever a single segment of a file is updated, this update in a leaf propagates through the whole *Chameleon Authentication Tree* and leads to an update of the root node, which serves as the public key for the verification of an individual file. In case the root node is further integrated into a *Revocation CAT*, this propagates again through the whole tree and leads to the noticeable event of an updated public key for all files. Since the protection of the root node for both the individual CAT and the *Revocation CAT* is based on a signature, an attacker cannot manipulate the CATs and thereby the revocation mechanism. In contrast to the implicit revocation mechanism of *VDSECC*, *RSA-Chaining* uses an explicit revocation mechanism based on the *WT-CRLs*: every revoked segment is added to the *WT-CRL* of the corresponding public key for the verification. If an attacker wants to manipulate the *WT-CRLs* that are stored at the PKI, this would require the

[5] HTTP is a stateless protocol, but many applications required state information, for example, when users should be able to log in to accounts. Session identifiers that are either incorporated to the URI, or stored in a so-called session cookie provide the required state information.

attacker to compromise the PKI, which is impossible according to our assumptions.

2.8.3 Active Attacker against CDNs, Web Caches, or WebTrust Content Servers

Finally, WebTrust also protects against active attackers that try to compromise the servers of a content distribution network (CDN), Web Caches, or WebTrust Content Servers. All these systems have in common that they simply serve already protected content to clients. So if an attacker manages to exploit a vulnerability in one of these systems, the only possible attack is to perform a denial of service. Since the attacker is not able to access the secret key for signing content, it is impossible for the attacker to replace or manipulate content without breaking the signatures, which would be easy to detect.

The situation slightly changes if the Content Server and the WebTrust Content Generator share the same system: if this server is compromised, secret keys still cannot be accessed by an attacker, but the system can be used as a signing oracle. In this case, malicious content can be validly signed without any chance for clients to notice during the verification. Therefore, we do not recommend this setting. However, if this setup cannot be avoided, one can use runtime attestation mechanisms to improve the chances of detecting a server breach (cf. Moyer et al. [149]).

2.9 Experimental Evaluation

Key to the success of a comprehensive authenticity and integrity framework for HTTP is both its performance in real-world scenarios and its ease of use. We conducted a performance evaluation of our prototypical implementation of WebTrust both at the client-side and for the WebTrust Content Generator. Additionally, we evaluate the network overhead introduced by WebTrust and discuss usability aspects of the system.

2.9.1 Performance Evaluation

In the following we provide the detailed results of our performance evaluation of WebTrust, both for the client-side implementation and for the WebTrust Content Generator. We would like to point out that since our prototypical WebTrust implementation does not yet include any communication with the PKI, the processing of PKI information is also not yet included. This would of course influence our performance evaluation and needs to be taken into account for estimating the potential performance of a production-ready implementation of WebTrust.

Our client-side performance evaluation was conducted on a Dell Optiplex 9010 Workstation equipped with an Intel Core i7 CPU and 16 GB RAM. The server-side performance evaluation was conducted on a similar Dell Optiplex 9010 Workstation, but equipped with an Intel Core i7 CPU and 32 GB RAM and running our patched Apache Tomcat version. Both machines (client and server) were connected with a 1Gbit/s uplink and packets transmitted between our client and our server had to pass 11 hops in between. The security parameters for our signature providers were fixed during our

performance evaluation according to the current NIST recommendations [156]: for our RSA signatures we use a key size of 2048 bits for both signature providers. In order to achieve a comparable level of security for both our signature providers *VDSECC* and *RSA-Chaining*, we additionally had to choose a security parameter for the chameleon hash function that achieves a comparable level of security as the 2048 bits key size. The chameleon hash functions in the *Chameleon Authentication Tree* for *VDSECC* are based on elliptic curves and we decided to use the elliptic curve `P-224`. Furthermore, our prototypical implementation of WebTrust makes use of SHA-1 as collision-resistant hash function [119]. For a production release we will shift to the winner of the NIST SHA-3 competition, the KECCAK hash function [157].

The runtime efficiency of WebTrust and its network overhead highly depend on the chunk size used for transferring data. Each WebTrust protected chunk is send with its own signature so that larger chunk sizes lead to less signatures and, thereby, to a better ratio between transmitted bytes for content and signatures. The goal is to find an optimal trade-off between the verification frequency, which depends on the chunk size and thereby on the amount of signatures, and the actual performance.

But how is the impact for real world applications and what is a meaningful content verification frequency? During our performance evaluation, we experimented with chunk sizes of 8 KiB, 32 KiB, and 128 KiB for the performance of the dynamic signature creation. 128 KiB chunks lead, for example, to a verification frequency of *one second* for video streams with a standard definition quality and 2 Mbit/s data rate. One second should be a reasonable verification frequency for almost all video-streaming settings. In addition to these three different chunk sizes, we additionally evaluated the performance of the server in case the required signature was already available in a server-side cache, i.e., the server was not required to compute the signature again. Using the different chunk sizes and the caching option, we performed the actual measurements based on the Siege benchmark tool [205]: we downloaded a 40 KiB file 10,000 times and simulated 100 concurrent users that tried to access the server. The limit for this performance measurement in our special setup was solely given by the computational power of the server.

Our comparison of the maximum number of parallel server transactions possible with WebTrust, standard *HTTP*, and *HTTPS* (cf. Figure 2.12) shows that if a signature is already available in the cash, WebTrust achieves a comparable performance as *HTTP* and outperforms *HTTPS*. For reasonable chunk sizes of 128 KiB with RSA-Chaining, WebTrust still outperforms *HTTPS* by almost 20%. *VDSECC* is already for a chunk size of 128 KiB slower than HTTPS, but we are still working on optimizations and hope to also outperform HTTPS for a 128 KiB chunk size in future.

Besides the scalability of the WebTrust Content Generator, we performed a microbenchmark for the client-side verification and we evaluated the full round-trip time for requesting a 40KiB document from the WebTrust Content Generator[6]. The client-side verification for a *VDSECC* signature takes on average $371\mu s$ (time for verifying one CAT node), whereas the verification of one RSA signature requires $121\mu s$. We would

[6]In this case the Content Server and the WebTrust Content Generator are the same entity: The signature is created on the server that also finally delivers both content and WebTrust signature to the client.

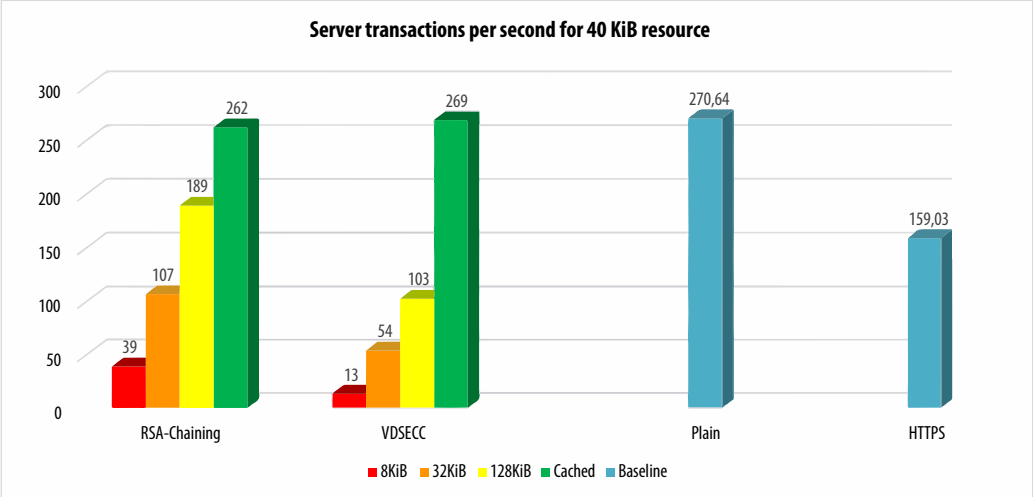


Figure 2.12: Comparison of server transactions per second for *VDSECC/RSA-Chaining* using different chunk sizes, HTTP and HTTPS (P7)

like to emphasize at this point that the RSA implementation is a widely used and highly optimized production implementation, whereas *VDSECC* was based on our prototype implementation.

Regarding the round-trip time evaluation for fetching a 40 KiB file from the content generator, our evaluation provides several interesting insights (cf. Figure 2.13 for an overview of the round-trip results): In case server-side caching is enabled[7], WebTrust achieves with both signature providers a round-trip performance comparable to *HTTP* and faster than *HTTPS*. Even if we include the client-side verification time, WebTrust is faster than *HTTPS* and only slightly slower than *HTTP*. Without caching, we can clearly see the impact of the server-side calculations of the signatures. The more signatures are required, the higher also the round-trip time. Nevertheless, even without caching, WebTrust with *RSA-Chaining* and using a key size of 2048 bits is still able to outperform *HTTPS* when using Diffie-Hellman key exchange and AES-256-CBC encryption for a 128 KiB chunk size. We still work on improving the performance especially of the *VDSECC* signature provider and aim at achieving also for this signature provider and a chunk size of 128 KiB a performance comparable or even slightly faster than *HTTPS*. But even with these improvements, *VDSECC* is still the only primitive with constant size revocation data. For a production implementation with frequent content updates this is highly desirable (for details on the revocation, cf. Section 2.6.3.3).

Besides the computational overhead, WebTrust also introduces a small network overhead when the WebTrust signature information is added to the individual segments. The size of the overhead depends, of course, on the signature provider used and its individual settings such as the security parameter. For the *RSA-Chaining* signature provider we use a key size of 2048 for the RSA signatures, which leads to a signature size of 344 bytes (transfer encoded based on Base64 [120]). The size of the signature information for *VDSECC* is about 167 bytes. Regarding the space overhead for the

[7] Caching is not possible for all scenarios, e.g., if content changes with every request.

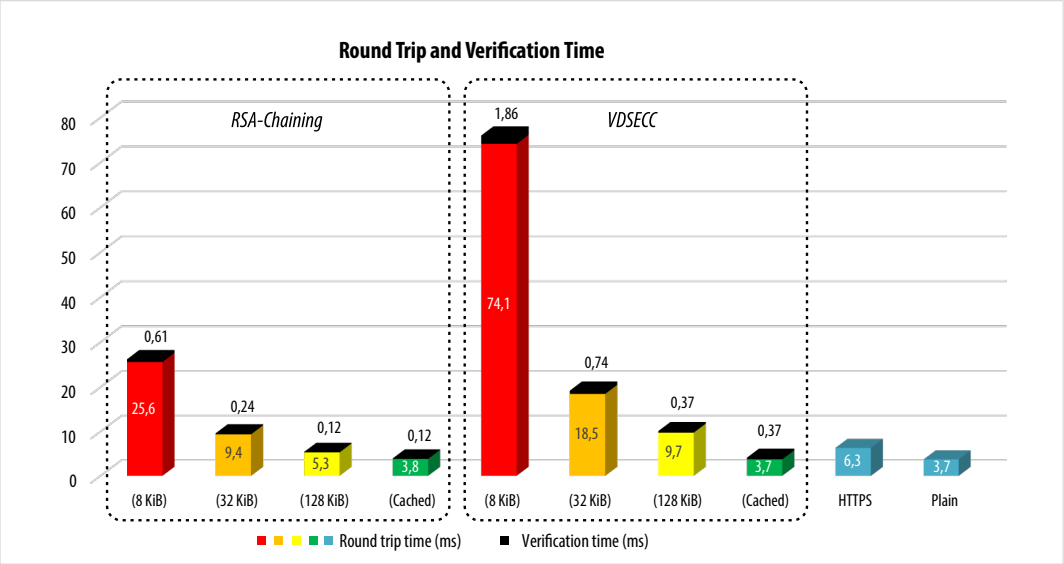


Figure 2.13: Comparison of the round trip time for requesting a 40KiB resource using VDSECC and RSA-Chaining in combination with different chunk sizes, HTTP, and HTTPS (P7)

network transmission, we consider our evaluation setting with small chunks of 8 KiB size as the worst-case setting, which leads to an overhead of 4% for signatures based on *RSA-Chaining*, and to an overhead of 2% for *VDSECC*. For our preferred chunk size of 128 KiB, which is even reasonable for most video streams, the overhead cuts down to 0.3% for *RSA-Chaining*, and to about 0.1% for *VDSECC*.

2.9.2 Usability Evaluation

Another central aspect for the acceptance of WebTrust for a comprehensive authenticity and integrity framework is ease of use. Our prototypical implementation integrates seamlessly into the chunked transfer encoding of HTTP 1.1, which provides full backwards compatibility with existing client-side and server-side solutions. We verified that extensions to the HTTP 1.1 chunked mode are in general supported by all major Web browsers such as Apple Safari, Google Chrome, Microsoft Internet Explorer, and Mozilla Firefox, as well as by popular tools used for HTTP requests such as cURL, GNU Wget, or Java [P7]. So we do not expect any negative side effects with existing software solutions. The usability of WebTrust itself can be compared to the usability of *HTTPS*. WebTrust needs to present positive or negative verification results of content to users in a understandable manner. As known from HTTPS, there is room for improvement for laymen users and we see the UI as important aspect of our future work. In addition to the verification information for full pages, we added an additional information pane for users to show the results of our individual verifiability, which also follows the existing approaches for presenting HTTPS authentication information.

2.10 Discussion and Future Work

We see our WebTrust system as a required and meaningful extension to the existing Web standards *HTTP* and *HTTPS*. WebTrust allows clients to verify the authorship and integrity of content while still enabling Web caches and content distribution networks. Especially for pre-signed content, WebTrust allows a large scale virtualization of the content server in combination with load balancers: the content servers do not have to store any keys and only need to take care of delivering the content using the extended chunked mode of HTTP. This is a major improvement in contrast to *HTTPS* in scenarios where only integrity is required. It will simply help to reduce the server and network load. Regarding the verification of content authorship, there is yet no integrative solution for HTTP at all. One could only use a naïve way of pure signatures that do not consider or reflect the typical use-cases on the Internet.

In order to achieve a production-ready implementation of our system, there is still work to be done for our Chromium setting: First of all, we need to implement the full PKI integration at the client side. Moreover, we want to move our mockup implementation of the individual verifiability to the browser itself. Although our signature providers already provide a reasonable performance for most settings on the Internet, we still see room for major improvements regarding the *VDSECC* performance, especially since we have not yet considered any multi-threading for our implementations. This would be required to support high-definition video and next generation video streams, which consume a much higher data rate than 2 Mbit/s. Depending on the particular use-case, one could also consider to pre-calculate keys for reducing the computational load, in particular at the client side [47, P7]. One could also consider an adaptive solution of WebTrust that allows to set chunk sizes scenario dependent, which can also lead to major performance improvements as our evaluation has shown. Finally, we envision also a direct support of WebTrust in content-capturing devices such as video cameras for live video streams. This would allow authors after a local authentication to directly create verifiable content.

2.11 Related Work

The standard protocol on the Internet for delivering Web content is the HTTP protocol [73]. But it was not designed to provide any security guarantees for connection. It neither provides authenticity or integrity, nor confidentiality. It does not even provide state information, as it is required for a user-individual communication on an account basis. To achieve a secured HTTP communication, the HTTP over TLS protocol (HTTPS [186]) is commonly used. But it only provides authenticated connection endpoints and an encrypted channel and does *not* provide authenticity with respect to the author or integrity for the document transmission between the author and receiving users. Depending on the usage scenario, users require different protection mechanisms and security guarantees and WebTrust aims at a comprehensive authenticity and integrity framework that coexists with HTTPS. WebTrust aims at allowing both the verification of authorship and the correct transmission of content from its author to any potential client. But WebTrust does not only coexist with HTTPS, it uses HTTPS whenever it is required to also provide confidentiality.

Table 2.1: Features required by a comprehensive authenticity and integrity framework for HTTP: a comparison of WebTrust and related approaches

Feature	1	2	3	4	5	6	7	8	9
SHTTP [187]	–	–	–	✓	–	–	S/D/–	–	–
HTTPS [186]	–	–	–	–	–	–	S/D/L	–	–
SSL Splitting [128, 129]	–	–	–	–	–	✓	S/D/L	–	–
Bayardo and Sorensen [26]	✓	✓	–	✓	–	✓	S/D/–	✓	–
SINE [79]	–	–	–	✓	–	✓	S/D/–	✓	–
HTTPI [55]	–	–	–	–	–	✓	S/D/L	–	–
Spork [148, 149]	–	–	–	✓	–	✓	S/D/–	✓	–
HTTPi [206]	–	–	–	✓	–	✓	S/D/L	✓	–
iHTTP [86]	–	–	–	✓	–	✓	S/–/–	✓	–
WebTrust [P7]	✓	✓	✓	✓	✓	✓	S/D/L	✓	✓

Legend: 1. Verifiable Authorship 2. Full Integrity 3. Document Revocation
4. Non Repudiation 5. Data Updates 6. Caching/CDN Support
7. Content types (**S**tatic, **D**ynamic, **L**ive Streaming,)
8. Progressive Verification 9. Individual Verifiability
✓: yes/full support, – no/no support

In the following we will compare WebTrust to related work and show where existing systems fail to close the current technology gap next to HTTPS. Before we start with the actual feature comparison of WebTrust and its closely related approaches, we would first like to introduce a scheme for signing digital streams by Gennaro and Rohatgi [83] from 1997. The authors propose to split streams into smaller blocks and most of the approaches presented in the following follow a similar paradigm. In case a stream is known in advance, for example a large video file that already exists, the authors split this file into blocks $b_1 \dots b_n$ and start to construct the authenticator beginning with the last block and ending with the first block. The authenticator A_i consists of the block b_i and the hash of the next block $\mathtt{H}(A_{i+1})$. The first authenticator A_0 provides the security guarantees for all following values since it includes a signature of the hash of (A_1,n). Creating the authenticator this way allows also the progressive verification of data; however, it does not support real-time streams since the full file needs to be known in advance. To cope with real-time streams the authors also introduce a scheme that uses a forward creation of the authenticator. The paper of Gennaro and Rohatgi does not consider the integration of the signature scheme for streams and the like in the Web context; this was done by the papers that are presented in the following.

We provide an overview of all features we consider relevant for a comprehensive authenticity and integrity framework and compare WebTrust with all related approaches in Table 2.1. As one can already see from the table, none of the related approaches supports all features as required by a comprehensive solution. During our feature comparison we considered a system to support a particular feature if it is supported in one of the introduced settings[8]. In particular, we consider the following features

[8]Some approaches introduce two or more different solutions (e.g., [79] or [206]

as mandatory: First of all, the framework requires (1) the possibility to verify the authorship of a document and not as for HTTPS the identity of the server that delivers the document. Next, it should be possible (2) for users to verify that a document was delivered without any modifications from the alleged author to them. Of similar importance is a mechanism (3) that allows the authors of documents to revoke them, for example, once information got invalid or changed, or simply an error was found, authors can invalidate an signed document such that users receiving the outdated document can notice. For many use-cases it is also important that users (4) can prove the correctness and authorship of a document to a third party, this feature called non-repudiation needs to be supported as well. In addition, authors should be able (5) to update protected document without the need of signing again the full dataset and a comprehensive framework should (6) support the caching of content at Web servers and by content distribution networks without rendering a later verification of these documents infeasible. Furthermore, it is very important that a comprehensive solution (7) also supports all relevant content types (i.e., static, dynamic, and real-time streamed content). It should be possible to (8) verify the correctness of a document while it is still in transmission. Such a progressive verification should allow clients to verify every incoming segment, which is crucial for the on-the-fly verification of live-streamed content. Finally, a comprehensive solution should support the (9) individual verifiability to provide guarantees for individual posts on blog-like Web sites. We do not list confidentiality as a desired feature since we consider it an orthogonal goal that can be accomplished by using WebTrust over HTTPS. Moreover, using an encrypted channel with individual keys per users, as done by HTTPS, prevents caching.

The first approach related to WebTrust that is listed in Table 2.1 is called SHTTP and stems from 1999 [187]. SHTTP is defined in RFC 2660 and has similar goals as HTTPS: It provides data confidentiality as well as an authenticity and integrity protection for data. But the authenticity and integrity protection is only provided for the connection between a client and the server, which does not fulfill the requirements we stated for WebTrust. This way, the approach also cannot support the *Individual Verifiability* of content. In contrast to the session-based HTTPS, SHTTP provides a message-based protection. The guarantees and features provided by SHTTP highly depend on the mode of operation chosen: it supports different key-exchange mechanisms, asymmetric and symmetric cryptography depending on the properties that should be achieved, and it can, but does not have to leverage a public key infrastructure. It may support non-repudiation depending on the scheme used, however, HTTPS is intended for protecting a client server connection. HTTPS does not specify a particular document revocation mechanism, it may only revoke the signing key used, which would invalidate all documents signed with that key and require all other documents to be signed again. The alternative would be to sign every file with a distinct key, which would also be very inefficient. To achieve a practical solution SHTTP would require an efficient revocation mechanism as used by WebTrust and support for data updates as well as progressive content verification.

HTTP over TLS, which is commonly referred to as HTTPS is defined in RFC2818 [186]. It follows as session based approach and provides a transport encryption for HTTP to

achieve data confidentiality for all static, dynamic as well as real-time streamed content[9]. It also provides a mutual authentication of communication partners based on certificates (optional on the client-side, the server-side certificate does not have to be part of a public key infrastructure.). The approach does neither provide a possibility to verify the authorship of documents, which also prevents the *Individual verifiability*, nor does it allow to guarantee the integrity of documents for the transmission from the content author to the person wanting to verify the integrity. Since HTTPS is session- and not message-based, it also does not provide non-repudiation or a document revocation mechanism. The use of session-individual keys prevents the caching of data and content distribution networks are not supported. The concept of data updates as introduced during our requirements analysis also does not really fit to the scenario of HTTPS: HTTPS uses symmetric encryption with a session key and encrypts everything that is transmitted in an individual session also with the corresponding session-individual key.[10]

SSL splitting [128] is an approach that aims at providing authenticity and integrity for data provided through Web caches in order to reduce the load/network congestion on the Internet. As the name already indicates the approach is based on SSL and the idea presented in the paper is to use a so-called *SSL splitting* proxy to merge authentication records from a Web server with data records from a Web cache to achieve server authenticity at the cost of loosing data confidentiality for the connection. But confidentiality is also not their goal. The authors make use of an SSL cipher suite that only provides authentication of the Web server and integrity for the data delivered via a Web cache: the `SSL_RSA_WITH_NULL_SHA` cipher suite. The whole approach is implemented as a patch to the OpenSSL library and their tests indicate that *SSL splitting* can reduce the network consumption by 25% to 90%. Their tests also indicate that for un-cached documents the latency deviates at most 5% from direct connections. Since the SSL handshake is still required, the computational load at the server is not reduced in comparison to HTTPS. An open problem is to understandably inform the user about the fact that a cipher suite not providing data confidentiality is used. Overall, *SSL splitting* achieves/misses from our required features the same features as HTTPS, but adds caching support at the cost of losing confidentiality in general (cf. Table 2.1).

Bayardo and Sorensen [26] propose the usage of Merkle trees [144] to authenticate Web responses. The tree is constructed as presented in Section 2.5 in Figure 2.2 and the signed root node is used to authenticate the content. But the root node is only known when the full content has already been created, so their approach cannot cope with real-time streaming. An adaption would be to continuously extend the tree when new data is generated so that packets can be delivered immediately. This would require to only sign and send the root node at the end when a stream has ended. However, this mechanism would prevent the progressive verification of content as required for real-time applications. Updates of single leafs in the tree are not supported and require the re-creation of the full tree. The system by Bayardo and Sorenson does also not

[9] The standard includes a so-called null cipher suite as well as weak cipher suites that should no longer be used.

[10] Not every data delivered in such connections is confidential, so caching could still make sense. It also depends on the required guarantees.

integrate a feature similar to our *Individual Verifiability*. Interesting is their idea to use DNS-SEC for delivering the root nodes of the Merkle trees for the later verification.

The *SINE* family of protocols [79] builds up on [83] and aims, similar to SSL splitting, at achieving a Web cache-friendly integrity protection for documents on the Internet. In contrast to SSL splitting, the retrieval of documents does not have to involve both the Web cache and the Web server. This reduces the load at the Web server since the remaining TLS handshake with the Web server that is still required by SSL splitting is no longer necessary.

SINE provides three protocol variants: the basic *SINE* protocol, the *SINEB* protocol, as well as the *SINEX* protocol. *SINE* splits documents in blocks and hashes them recursively from the last segment to the first segment. The final value of the recursive hashes gets concatenated with the number of segments of the document, a timestamp, and an expiration date. Subsequently, the concatenated values becomes hashed as well as signed and the result constitutes the so-called *Authentication Tag*. Although the *SINE* protocol does neither aim at the verification of the authorship of documents, nor at a proof that a document was correctly transferred from its author to any recipient[11], *SINE* could still be adapted to achieve this. *SINE* allows in general for progressive verification of arriving content, however, the *SINE* protocol does not support real-time streamed content. Also, *SINE* does not integrate the *Individual Verifiability*. The security guarantees provided by *SINE* are based on signing the final value of the recursive hashes, which supports non-repudiation, but the system does not provide a particular document revocation mechanism. The protocol itself is very efficient: it outperforms HTTPS and requires only a hash per segment of a document and one signature per document.

SINEB differs from the basic *SINE* protocol that way that the authentication tag is not signed during the creation. The remaining parts of the authentication tag creation are kept as well. The idea of *SINEB* is to reduce the amount of public key signatures/verifications if multiple pages are fetched from the same Web server during a single session. The authentication tag is communicated over a protected channel via HTTPs whereas the actual data is sent via HTTP. In comparison to the basic *SINE* protocol the verifiable authorship and the integrity guarantee from the author to any client is lost. The same holds true for the support of non-repudiation, it is also lost.

The idea of *SINEX* is to achieve the flexibility of the basic *SINE* protocol, but to still achieve a reduction of the signing and verification operations in case multiple Web sites are requested from the same Web server. Let us assume a client wants to receive the Web site w and this Web site links further pages $l_1 \dots l_n$. The authentication tags for $l_1 \dots l_n$ are now created as for *SINEB*, i.e., the tags are not signed at the end. The creation of the authentication tag creation for w includes now the unsigned authentication tags for $l_1 \dots l_n$ and the tag for w gets signed at the end. Therefore, the security guarantees can also be provided for all authentication tags for the pages for $l_1 \dots l_n$. The features supported by *SINEX* are the same as for *SINE*, a major difference can only be recognized for the performance: the more linked subpages of w are requested by a client during the same session, the more public key operations for signing and verification can be saved.

[11] We assume that the content author and the Web server are different entities.

The HTTP with Integrity protocol, short HTTPI [55], aims at providing message integrity and server authenticity. At the same time it aims at supporting Web caches. So it basically provides the same functionality as SSL splitting, including the requirement that Web caches need modifications to support HTTPI. The protocol is session based and claims to use a TLS-like protocol for establishing a symmetric session key. The key idea behind the HTTPI protocol is to decouple message headers and message content to achieve the Web cache support, although following in principle a similar concept as HTTPS does when using a `NULL_CIPHER` suite.

Moyer et al. present a system that stands out from the other approaches in our feature comparison: the *Spork* system [148]. In contrast to the previously described approaches, *Spork* integrates an integrity attestation mechanism for the Web server to provide guarantees that the server has not been compromised. However, it is in general very difficult to provide such guarantees: The integrity attestation mechanism leverages a Trusted Platform Module (TPM) and monitors the healthy state of system binaries, which does not defend against runtime attacks such as return oriented programming [201, 33, 50]. Another problem with that technique is the performance when dynamic or live-streamed content should be served with such guarantees. The creation of the attestation proof (based on the TPM `quote` operation) consumes run time (900 milliseconds)[12] and the network overhead can be quite huge: the authors' evaluation of *Spork* reveals a overhead of more than 85% for a 10 KB file, and still more than 65% for a 25 KB file. The implementation at the server side is integrated to the Apache Web server, whereas the client-side implementation leverages the Mozilla Firefox Web browser.

Spork's data protection mechanism itself follows the approach proposed by Bayardo and Sorensen and supports, therefore, both non-repudiation and progressive content verification; but the authorship as well as the correct transmission of documents from the author to any client cannot be verified and real-time streamed documents as well as the *Individual Verifiability* are also not supported. *Spork* does not integrate a particular document revocation mechanism, however, the *Spork* system is able to achieve freshness based on a trusted time server.

An extended version of the *Spork* system [149] adds an additional scenario where a database back-end provides content to the Web server for the creation of server-side dynamic Web sites. Similar to the Web server, it also includes a TPM and provides system integrity attestation. The evaluation of the extended *Spork* system reveals that using a dedicated server as database back-end reduces the throughput for dynamically generated Web sites at the Web server by roughly 20%; this overhead is already present without yet considering a potential overhead introduced by the additional integrity proofs. For the straightforward approach of using individual proofs per data object from the database server, the performance of the extended *Spork* system decreases dramatically. However, by using different optimizations and a different granularity for the proofs (not per data object, but one proof for a whole page, which includes also

[12] The authors perform optimizations for static pages which achieves a reasonable performance. But for dynamic pages Spork requires these 900 milliseconds. Depending on the network latency, this delay can be tolerated for many scenarios since the round trip time starting with the client request until the requested data is delivered is anyway dominated by the network latency. But if the network latency is low, people will probably not tolerate such a delay.

the proof for included objects), the performance is only slightly below the initial *Spork* system without a database back-end. Regarding our feature comparison there is no difference for the supported features between the extended *Spork* system [149] and the initial *Spork* system [148]. This idea of splitting the database server and the Web server is similar to our considerations of splitting the WebTrust Content Generator and the content server. However, we focus on authorship, which is especially for static content different to the model of *Spork*. In general, one could consider combining the approach followed by *Spork* for the integrity attestation with our WebTrust system, especially in the scenario where the WebTrust Content Generator and the content server are the same entity.

Another approach presented by Singh et al. is called *HTTPi* [206]. It aims at providing end-to-end transmission integrity and authenticity between a server and a client. However, it neither allows the verification of authorship (also not the *Individual Verifiability*), nor the verification of a correct transmission of documents from the author to any potential user. The approach works at a different level of granularity than WebTrust: *HTTPi* splits up chunks (as provided by the HTTP 1.1 chunked mode) into smaller segments and it still supports Web caches and CDNs without any further adaptation. We don't see this requirement for most of today's connections in combination with reasonable chunk sizes; however, this could be easily adapted by WebTrust. *HTTPi* allows the progressive verification of content, but it does not provide an option to securely update content. Content updates would either require a document revocation mechanism or a different noticeable event such as a change of the public key, since otherwise an attack could always replay an old version of content with a client being unable to notice. The time stamp included for *HTTPi* is at the moment required to be the same for all chunks of one resource to prevent replay attacks. However, this would require signing all data again if only one packet is changed. *HTTPi* was implemented for Microsoft's IIS 7.0 Web server and at the client-side for Internet Explorer 8. The performance evaluation was conducted based on a static Web site with a modified version of the Fiddler Web proxy and the results for *HTTPi* have been compared to HTTPS. On the one hand the end-to-end response time of *HTTPi* (without caching) is slightly worse than for HTTPS. On the other hand, the tested Web servers are able to handle almost as many *HTTPi* responses per second as for HTTP which is a clear improvement in comparison to HTTPS.

The final system in our feature comparison is *iHTTP* by Gionta et al. [86]. Their system aims at providing authenticity and integrity for HTTP response data from a Web server to clients without introducing a major performance overhead. So *iHTTP* has also no focus on the actual authorship of documents and misses the *Individual Verifiability*. The creation of their authenticator builds up on the concepts introduced by *HTTPi* [206] and SINE [79] and provides non-repudiation as well as progressive content verification. But similar to *HTTPi* and *SINE*, *iHTTP* does also not integrate a secure update mechanism as well as an efficient content revocation mechanism. The authors consider the Web server in the *iHTTP* setting as a trusted entity and focus on data that is delivered to clients without changes[13]. The scheme presented by Gionta et al.

[13]Gionta et al. refer to this data as "client-static data" in contrast to "client-unique data" that causes individual responses per client

does not support real-time streamed data, since the authenticator includes the length of the full HTTP response, but this can be solved. *iHTTP* provides freshness based on the newly introduced sliding window approach and supports Web caches as well as content distribution networks. The most interesting feature introduced by *iHTTP* is the so-called "opportunistic hash verification", which has the same intention as the *SINEX* variant from the *SINE* family of protocols [79]. The system includes in the DOM-element of a linked resource also the hash of the authenticator for the linked resource.

As a summary of the feature comparison between all closely related work and WebTrust we can conclude that the closest work to WebTrust is *HTTPi* by Singh et al [206]. However, their approach focuses on protecting the authenticity between a client and the Web server, without considering the actual author of content as data origin. Moreover, it does not include a data update mechanism and a meaningful data revocation. Two promising features of related systems for a future integration are the integrity attestation mechanism as presented by the *Spork* system as well as the opportunistic hash verification as presented by Gionta et al. [86].

Besides the closely related work discussed so far, there is also some interesting work with a different focus than WebTrust. The less closely related work either achieves only a really small subset of the features desired for a comprehensive authenticity and integrity framework (which these approaches do not aim for) as a byproduct when achieving their goals, or these systems do not provide any security guarantees for their protection.

The first less closely related system is *HTTPa*. *HTTPa* is defined in RFC 2617 [77] and describes a client-authentication scheme that can be considered as an improvement to the basic authentication scheme as defined in HTTP/1.0 [29]. This scheme is not designed to provide authenticity for delivered content, especially not with respect to the original author of content. However, we list it in the context of WebTrust since it allows protecting content sent from the server to the client. To achieve this, the client needs to request the `auth-int` mode and, subsequently, the server provides, based on *HTTPa*, an integrity protection for the full response. The integrity verification cannot be processed progressively such that the client-side has to wait until the full response has arrived. It is important to note that *HTTPa* does not provide any security guarantees for the authentication or the integrity itself. In order to achieve such guarantees the protocol needs to be used, for example, in conjunction with HTTPS.

Reis et al. introduce *Web Tripwires* to detect manipulations or changes of Web responses during their transmission over the network or, as they call it: "in-flight modifications of data" [185]. *Web Tripwires* consist of small pieces of JavaScript code that are embedded to Web sites; they can be evaluated once data has arrived at the client side. Adding only a small piece of JavaScript code at the server side offers, of course, a good performance, the same holds true for this lightweight kind of integrity check at the client side. The approach offers no protection against active attackers, since nobody would recognize when the JavaScript code is either modified or completely removed. The authors use their approach to evaluate how many Web pages are actually altered during the transmission from a Web server to a client (under the assumption that nobody actively manipulated the embedded *Web Tripwires*). Their findings show

that more than 1% of all received pages were changed: typical modifications they found included the injection of advertisements by an Internet Service Provider (ISP), content compression by ISPs, filters at the client side to remove advertisements or popups, as well as malware injected by attackers. But their evaluation analyzed such changes only for user requests on their own page. It would be very interesting to see how this happens for bigger Web sites that are, for example, commonly requested by mobile devices where Internet Service Providers (ISP) have a big interest in optimizing the network load. Especially, since using such a data-type specific compression to reduce the network load was already recommended by Fox and Brewer in 1996 [75]: they propose, for example, to reduce the resolution and color depth of images as well as to apply an additional compression to such images.

Vratonjic et al. analyze the integrity of Web content with a focus on online advertisements and propose, based on their findings, a new protocol for serving online advertisements, the *Data Integrity in Advertising Services Protocol (DIASP)* [230]. Their analysis shows that attackers perform on the fly replacements of ads with the intent to gain ad revenue for own ads, or with the intent of an attack. The authors also include an assessment of the potential economic impact of such attacks. As a typical scenario for such attacks the authors consider either clients that enter the access point of attackers or (free) ISPs where ad injection might be part of the business model. Their solution called DIASP introduces an authenticator based on a hash chain that is signed by the advertising service. We consider it difficult to integrate a separate authenticity and integrity mechanism for online ads, especially since this also requires a meaningful presentation of the result to the users. We would follow in their case an approach related to our approach for the *Individual Verifiability*. The main page would include already the WebTrust protection for the code that requests the advertisements with the correct identifiers for the revenue. So these identifiers or the advertisement service itself could no longer be altered. Later, when the advertisements have been requested from the advertisement server and were already locally rendered, the authenticity information for a particular advertisement should be displayed as for a blog example with the *Individual Verifiability* of the posts.

In contrast to the already presented papers that aimed either at achieving authenticity and integrity of Web content or at least at detecting when integrity of delivered documents is harmed, the following papers propose authentication mechanisms for particular use-cases (for example, special file formats or lossy channels).

Lin and Chang introduced already in 1998 a signature algorithm specifically tailored to JPEG-images and MPEG-videos [133]. The authors designed the signature procedure that way that it is able to distinguish content-changing manipulations from content-preserving manipulations [133], i.e., it should still verify for a compressed or resized image, but not if somebody removed something from the image. In order to achieve, for example, a compressed but authentic JPEG-image, the authors exploit the fact that the compression step relates two JPEG coefficients to each other. This allows proving within some bounds (compression is lossy), that an image was simply achieved by compression. To counter imprecision of optimized libraries such as libJPEG, their approach uses an additional error bound that is tolerated. The authors exploit specifics of different data formats to predict within some bounds a particular transformation of a

formerly known authentic image. As long as the transformation is consistently applied, manipulation could only have been applied in form of the transformation itself.

DOMHASH [137] is defined in RFC 2803 and aims at providing unambiguous hashes of nodes for XML documents at the level of the Document Object Model (DOM). It targets at two different usage scenarios: first of all the included hashes can form the basis for signing the document later on to perform authenticity and integrity checks; second, it should form the basis for synchronizing two different DOM structures. Working at the DOM level helps to prevent ambiguities since the same DOM structure can be stored in different files (for example due to different white spaces, line breaks, character encodings, etc.). The *DOMHASH* approach could, for example, be used to implement our *Individual Verifiability* at a different level of granularity: at the XML level instead of at the level of `iFrames`. *DOMHASH* could also be integrated into the opportunistic hash verification as presented by Gionta et al. [86].

Another scheme that aims at authenticating XML documents is presented by Devanbu et al. [62]. Their focus is on authenticating answers to selective queries over XML documents while assuming an untrusted Web server. The key idea behind this scenario is similar to our WebTrust scenario since the author of content is the person who provides the initial proof of integrity and authorship. In order to verify the answer to a query, both the Web server for answering the query as well as the author for proving a proof are involved. Starting with a naïve approach based on the previously introduced DOMHASH [137], Devanbu et al. introduce a new data structure called *xtrie* to achieve a more efficient evaluation. Similar to our intent with WebTrust, Devanbu et al. also want to prevent that the online Web server has the private key for signing data, since it would get exposed in case the server gets compromised. In addition, the authors also want to prevent insider attacks. In comparison to WebTrust the authors focus at a different level of granularity. They can certify all possible answers to queries that are valid according to a valid XML document. But the authors do not only aim at providing integrity and authenticity of an answer to a query, but at completeness.

In 2004, Ray and Kim present a collective signature for the authentication of XML documents. The authors assume static XML documents and their system is based on one way accumulators. The major idea of the paper is to provide a primitive that allows an easy authentication of single XML sub documents. For larger documents, the approach introduces quite some overhead since every data node in the XML document is signed upfront. In case of static XML documents there are valid usage scenarios; however, for dynamic documents, for example, in the Web setting, the overhead is too much.

Finally, Perrig et al. present two schemes for signing and authenticating multicast streams over lossy channels [170]. The first scheme is called *Timed Efficient Stream Loss-tolerant Authentication* (TESLA) and it is based on a commitment scheme and message authentication codes. The system is initially bootstrapped by signing the first packet. The second scheme is called *Efficient Multi-chained Stream Signature* (EMSS) and it is based on digital signatures. It provides in contrast to TESLA non-repudiation of each individual packet and it does not need a strong time synchronization. In the basic EMSS scheme packets contain hashes for previous and successor packets and they are frequently interleaved with so-called signature packets for the last few hashes. The

authors analyze for EMSS different hash concatenation configurations to achieve an optimal trade-off between the overhead and the actual loss tolerance. In contrast to WebTrust, EMSS potentially introduces a verification delay for clients (depending on the frequency of the signature packets), which would hinder a progressive on-the-fly verification of incoming content. Our CAT-based *VDSECC* protocol has here clearly an advantage for the verification, since all required information can be provided to the clients when needed.

3 X-pire!

Within the last decade, people got used to publishing various kinds of information on the Internet and this data commonly also includes highly personal information. Especially the success of social networks has been a huge catalyst for the amount of personal information on the Internet. In July 2009, a survey conducted by dimap on behalf of the German Federal Ministry of Food, Agriculture and Consumer Protection (BMELV) revealed that already about 30% of the employers use information on the Internet to evaluate candidates in hiring procedures [39]. Privacy concerns and the public discussion about personal information in the Internet led politics to publicly demand for a mechanism that allows to digitally wipe-out previously published data from the Internet [38]. This chapter presents X-pire! and X-pire 2.0, which are designed to implement an expiration date for digital data. In the following, we exemplify the utility of both systems on the use case of images in social networks and introduce the typical workflow of our protection mechanism. Before we discuss X-pire! and X-pire 2.0 in detail, we motivate both approaches and provide a verbose problem description. Major parts of the content in this section are taken from the author's publications [P1] (for Section 3.3) and [P8] (for Section 3.1, 3.2, 3.4, 3.5).

3.1 Motivation

Today, a huge amount of people around the world at all ages has accepted the social networks' free dissemination of personal information, since the pervasive availability of such published information provides us with great opportunities and utility. No matter whether we look at business focused social networks or at ones focusing on personal private networking, these systems allow us to efficiently connect people around the globe while providing great usability even for laymen users. The connectedness and liveness of published information lately got further amplified by the huge spreading of mobile devices, which allow us to communicate and publish information (for example,

information on social networks) anytime and anyplace.

However, today's information culture to acquire and instantly publish all kinds of data also causes severe privacy problems. Most of today's Internet users, especially teenagers, publish all kinds of potentially unflattering sensitive information, such as images, videos, or even text, without recognizing that this might destroy their future reputation. Especially young people do not recognize that the information they revealed is preserved and might be detrimental to their future life and career. Besides a better education that explains our today's technologies and its individual risks especially to children and teenagers, we also need to understand the root causes of todays privacy problems on the Internet and to address these current privacy problems with new privacy mechanisms. An important step in the right direction regarding the education of young people is the annually organized *Safer Internet Day* from the European *Insafe* network, which aims with its events and advertisement campaigns at the promotion of a "more responsible use of online technology and mobile phones, especially amongst children and young people across the world" [190]. But what is the root cause of the fact that people publish highly personal information on the Internet without recognizing the potential future impact of such information?

We think that this lack of recognition is caused by the users' wrong understanding of how the Internet functions, which also leads to the users' wrong expectation of how long their data is actually stored in the Internet. Their expectations about the lifetime of data on the Internet simply do not match the actual lifetime of data on the Internet. For most users, the Internet basically consists of what popular search engines present in their search results and people usually only look at the first result pages. They are simply not aware of the fact that caching in search engines or content duplication by mirrors and data aggregators lead to a virtually infinite memory of the Internet that can still be accessed. This information is just slightly more difficult to find. And although this search process is slightly more complex, people need to be aware of the fact that this process exists and can even be automated. The potential for automation is the major difference between searching information in digital archives and searching information in huge traditional paper archives. Providing an efficient search in huge paper archives is very difficult and expensive and in speed and precision not at all comparable to search engines on the Internet. Especially a large-scale retrieval of information was for classical archives close to impossible and this is what people still have in mind. This observation is also discussed by work of Mayer-Schönberger [142], who compares modern electronic storage in the age of information technology with the traditional archiving of paper documents. He concludes that the traditionally limited lifetime of data fuels the expectations of average users that digital data is subject to a similar expiration process. In short, people expect that digital data also expires and that they have a technical means to keep control over their published content.

To meet this expectation, the European Union recently included into their draft for the new *European General Data Protection Regulation* an explicit article that mandates the "right to be forgotten and to erasure" (Article 17 in [69]). In a similar direction goes a bill of the state of California in the United States that was signed in September 2013 and will become active on January 1st, 2015 [213, 93]. It focuses on protecting the privacy of minors under the age of 18 by forcing operators of commercial Web services

to provide information on how to delete postings of minors and also to actively support and enforce the deletion of postings afterwards [213, 93]. However, in comparison to the European efforts, this can only be seen as a first step, since it does neither include the deletion of replicated data nor the deletion of adults' postings. The most recent advance in the area of the right to be forgotten is the decision of Google to integrate a general feature to request the deletion of search results from Google's index [89] as a result of a lost trial in front of the Court of Justice of the European Union [112]. According to Bloomberg Google received already on the first day more than 12.000 requests for deletion [241].

3.2 Problem Description

So in order to meet the expectations of users, we face the technical challenge to imitate the traditional expiration of analog data (for example, of information written on paper such as classical newspapers or magazines): We need to develop a digital expiration date for digital data that is able to cope with the requirements given by today's information culture. On the one hand it needs to provide users with the desired privacy guarantees: after data has been published, people usually forget it after some time and this should not strike back. Data should simply vanish after some time automatically according to an expiration date. One the other hand, a digital expiration date mechanism requires a seamless integration into common user activities in the Internet, such as publishing and consuming digital content in online social networks or simply on commodity Web servers. Otherwise, a wide acceptance by laymen users will be out of reach. There have already been many attempts to solve this technical challenge [31, 168, 167, 152, 82, 46, 184]. All of these approaches follow a similar technical paradigm when trying to solve the challenge of meeting the users' expectations regarding the expiration: data to be protected is encrypted with a symmetric key and only the encrypted data is published. Afterwards, access to this published but protected data is controlled by restricting the access to the symmetric key required for decryption. Especially in cases where the publication platform is not trusted, this technical paradigm is the only solution possible.

However, the paradigm of encrypting data does not solve the full challenge of providing a usable digital expiration date. A full-blown solution needs to cope with publication platforms that do not intend to collaborate with such a solution. Online social networks such as Facebook or Flickr allow for example to upload images in certain data formats (for example, as JPEG-files), but they will not support, for example, to upload an encrypted file as an image. A practical solution needs to consider this as well. The protection needs to be integrated seamlessly into existing data formats so that they are still compatible with existing upload routines as provided, for example, by online social networks. We offer with X-pire! the first solution that provides a seamless integration for images in social networks while still following the technical paradigm of encrypting data and restricting the key access.

All mentioned approaches, including X-pire!, assume that legitimate users of the system follow a benign behavior. This is a common assumption that is used, for example, in the *Controlled Access Protection Profiles* by the National Security Agency of the United States of America following the Common Criteria standards [114]. From

our point of view, the particular setting of X-pire! poses also the technical limit of software-based solutions.

However, all mentioned approaches, and to a certain extend also X-pire!, suffer from one major shortcoming: none of them provides a solution to the so-called *data duplication problem* (DDP)[1], which means that before the expiration date has been reached, legitimate users and attackers can easily view, duplicate, and publish protected content. This, however, removes the expiration date from protected content. While one might still be willing to trust that images uploaded in social networks and only visible to selected trustworthy people (for example, friends or family) are not duplicated within a short time frame (if one additionally trusts the providers of social networks, ignores caching, etc.), this clearly does not hold true anymore if data becomes publicly available without any additional access control. Solving the DDP is central for any approach aiming at a comprehensive expiration date for digital content, since unprotected copies would break the system. The successor of X-pire!, X-pire 2.0, is the first comprehensive solution for a digital expiration date: it provides all major features of X-pire! and solves the DDP, thereby being the first solution that is able to provide strong security guarantees. In the following, we will first explain X-pire! and introduce afterwards its successor X-pire 2.0.

3.3 X-pire!

X-pire! constitutes the first solution for a digital expiration date for images that allows for a seamless integration into social networks such as Facebook [71], wer-kennt-wen [233], Google+, or Flickr[2] [74]. It provides users with a novel and flexible system for publishing images in social networks with an expiration date that integrates into the common publication workflow and does not require direct support or active collaboration of existing social networks. X-pire! follows, similar to related approaches, the paradigm of encrypting data to be protected and controlling afterwards the access to the required decryption key. However, it provides additionally a robust JPEG embedding as required for existing JPEG upload routines of publication platforms[3]. After images protected by X-pire! have been published, its owners can modify the expiration dates of images and even enforce the instantaneous expiration of data. Once the expiration date set by the user is reached, the published image becomes automatically unavailable. Although we provide with X-pire! a pure software solution that focuses on the publication of protected images on the Internet and in particular on social networks, we would like to emphasize that X-pire! provides a general concept for such a system that is not limited to a particular data type or application scenario. X-pire! targets two scenarios: 1) the publisher has control over the publication platform, for example, the own Web server,

[1] X-pire! provides a mitigation to this problem, but does not fully solve the data duplication problem, for a discussion of this limitation please refer to Section cf. 3.3.7

[2] At the time of its development, X-pire! supported Facebook. In the meantime, however, Facebook changed the JPEG compression method from baseline JPEG to progressive JPEG, which is currently not implemented by X-pire!. Wer-kennt-wen quit service on June 1st, 2014.

[3] There is no incentive for publication platforms to change their upload routines for our system, because our system is against their business model of acquiring data.

and 2) the publisher has no control over the publication platform, like in modern online social networks.

3.3.1 Contribution

X-pire! implements a digital expiration date for digital data, especially for images in social networks. In particular, X-pire! provides the following major contributions:

1. X-pire! allows publishers to control their published data by providing an expiration date for digital data, including instant expiration and expiration date management. In order to achieve the digital expiration date, we follow a paradigm that has been used similarly in previous approaches: digital content is encrypted, and access to it is controlled via the access to the decryption key. When content should become unavailable, access to the decryption key gets denied or the key gets deleted.

2. Since we protect content by encrypting it, we face compatibility problems with the existing infrastructure. Upload routines for JPEG images in social networks, for example, will only accept valid JPEG files, and even uploaded JPEGs will be rescaled and re-compressed. Solving these compatibility problems has also not been considered by previous approaches so far and X-pire! is first to provide a novel robust JPEG embedding for arbitrary data.

3. We show how our system seamlessly integrates into the existing infrastructure by illustrating how protected images can be published in social networks (for example, Google+ and Flickr) and on static Web sites. X-pire! integrates into users typical workflow when browsing the Web and provides a one-click solution.

3.3.2 System Overview

In the following, we provide a high-level overview of X-pire! and discuss both the technical challenges we faced as well as the systems' underlying prerequisites and assumptions. Moreover, we outline the attacker model for the system.

3.3.2.1 High-level View on the Protocol

The typical usage of X-pire! is split into three different phases: the so-called *Publication Phase*, the *Viewing Phase*, and the *Update phase*. The *Publication Phase* involves the so-called *X-pire!-Publisher* $\mathcal{P}$, the *X-pire!-Keyserver* $\mathcal{K}$, and a *Content Server* $\mathcal{C}$. During the *Viewing Phase*, $\mathcal{K}$ and $\mathcal{C}$ are involved as well, but the *X-pire!-Viewer* $\mathcal{V}$ replaces $\mathcal{P}$. For the *Update Phase* only the *X-pire!-Publisher* $\mathcal{P}$ and the *X-pire!-Keyserver* $\mathcal{K}$ are involved. A high-level overview of the full system is provided in Figure 3.1.

Publication Phase During the *Publication Phase*, the X-pire!-Publisher $\mathcal{P}$ first protects a previously captured photo/existing image by encrypting it with a key received from the X-pire!-Keyserver $\mathcal{K}$ and assigning an expiration date to it. Afterwards, the encrypted data is embedded into a container image that is compliant with the desired Content Server $\mathcal{C}$. Finally, $\mathcal{P}$ publishes the container on this desired Content Server $\mathcal{C}$ on the Internet, for example, on a public Web server or in a social network.

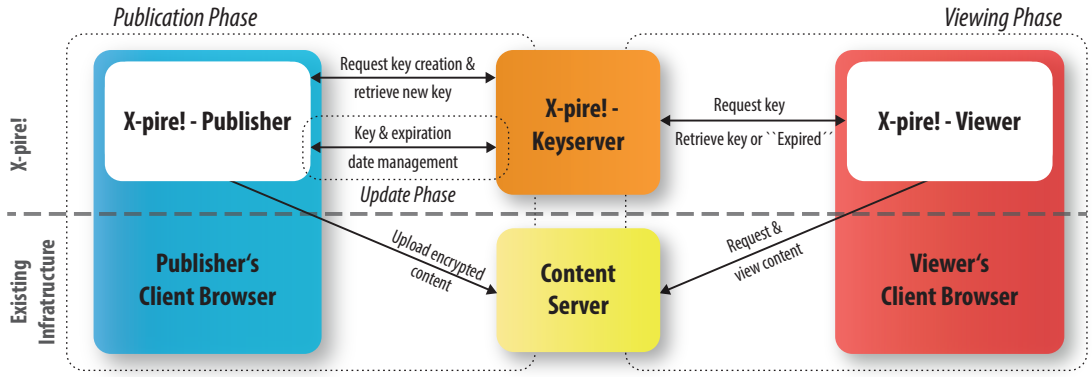


Figure 3.1: X-pire!: A high-level overview

Viewing Phase Once an image is available on a content server $\mathcal{C}$, X-pire!-Viewers can view protect images during the *Viewing Phase*. Therefore, a X-pire!-Viewer $\mathcal{V}$ can request the protected image from the content server $\mathcal{C}$ and the corresponding key for decryption from the X-pire!-Keyserver $\mathcal{K}$ to decrypt the protected image locally. Now the original can be viewed by the viewer $\mathcal{V}$.

Update Phase The *Update Phase* allows the X-pire!-Publisher to modify the expiration dates of already existing protected images by changing these properties for the decryption keys at the X-pire!-Keyserver $\mathcal{K}$. This includes the shortening and extension of already existing expiration dates up to an instantaneous expiration of protected images.

3.3.2.2 Technical Challenges

During the design of X-pire!, we needed to solve several technical challenges to achieve the goal of a digital expiration date for images with a seamless integration into the users' workflow when using social networks.

Achieving a Robust JPEG Embedding First and foremost, we needed to invent a robust embedding of encrypted data into JPEG images. JPEG is still the most common image format on the Internet (cf. Figure 3.2) and basically every publication platform that supports images also supports JPEG. Since we want to publish protected images on the same platforms we publish unprotected images on (for example, common online social networks), we have to embed encrypted images into JPEG-files in a way that follows the JPEG standard to retain compliance with the existing infrastructure. Enabling such an embedding for social networks, where typically most of the privacy sensitive images are published, requires X-pire! to cope with the typical re-encoding (including resizing and compression) of JPEG images as typically applied in post processing routines during the upload procedures of social networks. Thus, simply uploading the encrypted data c would not be possible, since c does not retain a valid JPEG as expected by the upload routine of a social network. But even if one solves this compliance challenge and

produces a document c that embeds an encrypted image and still retains a valid JPEG file, this results in the publication of a re-encoded version c', in which the compression has potentially destroyed the encryption. Nobody would now be able to decrypt the image embedded into c', which is similar to its instantaneous expiration. To solve this problem, we developed for X-pire! a novel technique for embedding encrypted information within JPEG files that follow the standard and where the embedded data survives JPEG compression without requiring any support from existing infrastructure. Achieving a robust solution to this problem was not only challenging in theory: the implementations of the JPEG standard used in the Internet are often highly optimized regarding the performance. In several cases, performance improvements are favored in contrast of precision, for example, by introducing rounding errors. This inaccuracy introduced by several implementations needs to be explicitly countered. We need to recover the encrypted image with 100% accuracy; otherwise we cannot decrypt the image. Our final solution that also addresses this inaccuracy still does not need any explicit support from existing Web servers or social networks, and thus allows for a seamless integration into the existing infrastructure.

Mitigation of the Data Duplication Problem Second, a major problem of all previously proposed solutions for a digital expiration date is that they do not consider the data duplication problem: If an attacker takes the role of a legitimate user, both content and keys can be accessed and copied during the *Viewing Phase* before the expiration date has been reached, which removes the X-pire!-based protection and thereby the expiration date. To the best of our knowledge, we are the first ones to consider this problem. In order to mitigate the data duplication problem, we decided to build X-pire! based on a dedicated keyserver. This allows us to introduce security measures at the server side to restrict the access to keys, which prevents the crawling of keys on a large scale and in an automated manner. Although the dedicated X-pire!-Keyserver introduces a potential weakness (the server constitutes a known address where everybody could request keys for decrypting X-pire!-protected content), the same mechanisms that mitigate the data duplication problem also mitigate this potential weakness.

Flexible Expiration Dates Third, we wanted to design a system that overcomes the limitations of related approaches regarding the flexibility of the expiration dates[4] and supports a fully flexible expiration date. Achieving this flexible expiration date was another reason for our design decision to base X-pire! on a dedicated keyserver. A dedicated keyserver allows us to fully control the storage environment for keys: It allows us to set arbitrary expiration dates and even to change the expiration date after protected images have already been published. X-pire!-Publishers can even go as far as to deploy their own keyserver to stay in full control over the storage environment without the need of trusting any infrastructure provider. In case publishers are still in favor of a distributed setting that leverages existing infrastructure to store keys (for

[4]Related approaches such as Vanish [82], EphPub [46], or the system presented by Reimann and Dürmuth [184] cannot directly control the decentralized environment that is used for storing the keys (DHT-based decentralized network, Web sites, DNS entries). Therefore, the time-frame for expiration is dictated by the infrastructure: in case of Vanish about 8 hours, or for EphPub at most 7 days.

example, as for Vanish or EphPub), we would like to mention that X-pire! is in principle still compatible with these distributed approaches.

Seamless Integration Finally, we wanted to provide a seamless integration of X-pire! into the typical workflow of users when browsing Web sites on the Internet and, in particular, when using social networks. Therefore, we needed to design both the X-pire!-Publisher and the X-pire!-Viewer to meet this requirement: all encryption/decryption and embedding/extraction routines should be integrate with as few user interaction as possible. Especially the viewing phase should be fully transparent to the user. As long as the expiration date of an image has not been reached, the image should be automatically decrypted and replace the protected version of the image on Web sites.

3.3.2.3 Requirements and Assumptions

In order to achieve the goals of X-pire!, the following requirements and assumptions need to be met:

- We assume the users of the X-pire!-Viewer to be honest and to neither create persistent copies of decryption keys nor decrypted content.
- We require a Captcha service to prevent automated crawling attempts.

3.3.2.4 Attacker/Threat Model

- We assume that an attacker can take the role of a viewer, however, we do not consider collaborating attackers.
- We assume that an attacker cannot compromise the X-pire!-Keyserver. Attackers can only query the X-pire!-Keyserver according to the X-pire! protocol for the *Publication Phase* and the *Viewing Phase*.
- We assume that an attacker is able to read, intercept, and modify arbitrary packets on the network. Attackers can only decrypt encrypted content if they legitimately received the required key according to the protocol.
- The attacker cannot compromise the publisher. However, he can take the role of a X-pire!-Publisher and protect and publish images.

3.3.3 JPEG Primer

Before we describe in detail how X-pire! is integrated into the JPEG workflow of compressing and decompressing images, we will first provide the necessary background information on the JPEG standard itself. The Joint Photographic Experts Group developed the standard already in 1992 and the JPEG format constitutes up to date the most common image compression technology on the Internet (cf. Figure 3.2). When we usually refer to JPEG images or files, we typically refer to a so-called JFIF file [99], the JPEG container format. The JFIF standard defines the structure of the file in which the actual JPEG compressed image data and corresponding meta information is stored.

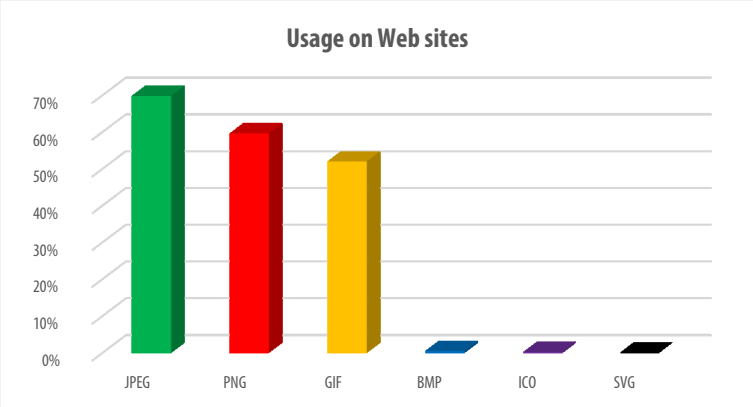


Figure 3.2: Statistic on common image file formats used on Web sites on the Internet. Web sites may use several images file formats at the same time. As of October 2013 (232)

Mode	Precision	Compression	Coding order	Coding type
Baseline DCT	8-bit	DCT	Sequential	Huffman coding
Extended DCT	8-bit or 12-bit	DCT	Sequential or Progressive	Huffman or Arithmetic coding
Lossless	2 . . . 16-bits	Predictive	Sequential	Huffman or Arithmetic coding
Hierarchical	2 . . . 16-bits	Predictive or DCT	Sequential or Progressive	Huffman or Arithmetic coding

Table 3.1: Comparison of JPEG modes of operation based on (122)

The image data itself can be treated in several ways. The JPEG standard introduces four different modes of operation: the *Baseline mode*, the *Extended DCT-based mode*, the *Lossless mode*, and the *Hierarchical mode* (cf. [122]. An overview of all JPEG modes is provided in Table 3.1). The two most common modes of operation for JPEG images on the Internet are the *Baseline DCT* mode and the progressive version of the *Extended DCT-based mode*, which from now on is referred to as the *Progressive mode*. Both modes of operation do not differ substantially regarding the different encoding steps, but merely in the ordering of the values that are input to the encoding routines [122, 166]. The advantage of the progressive mode is that it allows to render the full image step-wise, i.e., from a high-level representation to the final version with all details (for example frequency-wise, starting from low frequencies to high frequencies [166]). This allows, for example, Web browsers to already provide a full preview image while the image is still being loaded, which then becomes continuously "sharper". The Baseline DCT mode is structurally more simple since it does not apply this re-ordering, but loads images sequentially already with full details. For X-pire!, we focus on the *Baseline DCT* (BDCT) mode of operation, since it was used at the beginning of the X-pire! development (in 2010) used by all major social networks (for example, by Facebook and Flickr)[5]. However, our approach is in principle not limited to this mode.

In the following, we will describe the encoding process of JPEG images with the Baseline DCT mode of operation as shown in Figure 3.12. The corresponding decoding process consists of the inverse application of the mentioned steps (cf. the decoding

[5]Since the development of X-pire!, Facebook has moved on to the progressive mode of operation.

$$
\begin{aligned}
Y &= 0.299R + 0.587G + 0.114B \\
Cb &= -0.1687R - 0.3313G + 0.5B + 128 \\
Cr &= 0.5R - 0.4187G - 0.0813B + 128
\end{aligned}
$$

Figure 3.3: JPEG color conversion from RGB color space to YCbCr color space as defined in (99)

process in Figure 3.12). The JPEG primer is based on the definitions of the JPEG standard itself [122] and the detailed explanations by Pennebaker and Mitchell [166]; for more detailed information about the JPEG procedures we also refer to these references.

RGB to YCbCr Conversion At the beginning of the JPEG BDCT encoding, the image to be encoded is converted from the RGB color space model to the YCbCr color space model [166]. The starting image in the RGB color space consists of $x * y$ pixels (x in horizontal direction and y in vertical direction) and stores for every pixel at position (x, y) in the image a triple (R, G, B) where R is the "red" value, G is the "green" value, and B is the "blue" value. All values range in the interval from 0 to 255. The conversion to the YCbCr model follows the formula shown in Figure 3.3 and results in three different channels, the luminance channel Y providing brightness information as well as two chrominance channels that provide color information: the chrominance channel for the blue-difference Cb and the chrominance channel for the red-difference Cr [182].

After the color space conversion has finished, the color channels may optionally be subject to *subsampling* [122, 166, 103], which is usually only applied to the chrominance channels. The idea behind subsampling the chrominance channels is to exploit limitations of the human visual system to save space by discarding information that humans cannot perceive anyways [103]. Subsampling is usually defined for each color channel individually and both for the x and y directions. Having, for example, a subsampling factor of 2 for both the x and the y direction leads to a reduction of $x/2$ and $y/2$ values in the color channel, i.e., $1/4$ of the original information.

In case no subsampling is applied, or after subsampling has finished, the JPEG standard requires a level shift by subtracting 128 from all values [122, 166]. The image values range now from -128 to 127 instead of from 0 to 255. In a next step, all x by y channels are split into 8 by 8 blocks as shown in Figure 3.4. The blocks are generated row-wise from top to bottom and in each row from left to right. The 8 by 8 blocks that have been extracted in this preparatory step constitute now the input for the subsequent discrete cosine transform (DCT).

Discrete Cosine Transform The discrete cosine transform (DCT) is performed iteratively on all previously generated blocks. The 8 by 8 input blocks consist of `Integer` values and the DCT produces 8 by 8 output blocks with the so called DCT coefficients. Inside of each output block, the DCT coefficient at position $(0, 0)$ is referred to as DC coefficient; all other coefficients are referred to as AC coefficients (cf. [122]).

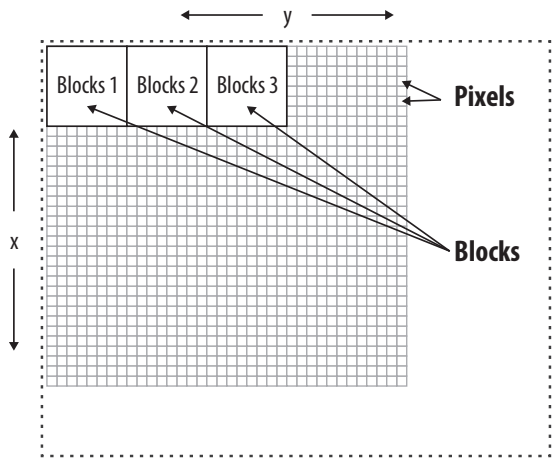


Figure 3.4: JPEG block extraction

$$
\begin{aligned}
\text{FDCT}: \; S_{vu} &= \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} \cos \frac{(2x+1)u\pi}{16} \; cos \frac{(2y+1)v\pi}{16} \\
\text{IDCT}: \; s_{yx} &= \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v S_{vu} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \\
C_u, C_v &= \frac{1}{\sqrt{2}} \quad \text{for} \quad u, v = 0 \\
C_u, C_v &= 1 \quad \text{otherwise}
\end{aligned}
$$

Figure 3.5: Discrete cosine transform as defined in Annex A of (122)

Quantization After the discrete cosine transform, the so-called quantization is applied. This step constitutes the actual compression step of the encoding process, i.e., this step is usually lossy. The quantization itself is applied by dividing each 8 by 8 block with the DCT coefficients position-wise by the so-called quantization matrix. A typical example of such a matrix is provided in Figure 3.6[6]. In case all values of the quantization matrix are 1, no compression is applied.

Huffman Encoding As a preparatory phase for the *Huffman encoding* [108], a differential encoding for the DC coefficients is applied and the 8 by 8 block is transformed into a one-dimensional sequence [122]. The processing order for the blocks is the same as for the block creation: it is conducted row-wise and in each row from left to right. To achieve the differential encoding, the DC value of the first block is kept as is. For all following DC values, the DC value of the previous block is subtracted from the current 8 by 8 block. This difference is now kept. To achieve the one-dimensional ordering, the values of the 8 by 8 matrix are added subsequently by choosing them in zigzag order [122]. The exact ordering is shown in Figure 3.7. The advantage of the zigzag

[6] At the time X-pire! was designed, Facebook used the quantization table that is shown in Figure 3.6.

5	3	3	5	7	12	15	18
4	4	4	6	8	17	18	17
4	4	5	7	12	17	21	17
4	5	7	9	15	26	24	19
5	7	11	17	20	33	31	23
7	11	17	19	24	31	34	28
15	19	23	26	31	36	36	30
22	28	29	29	34	30	31	30

Figure 3.6: Quantization table used by Facebook in 2010 (luminance channel)

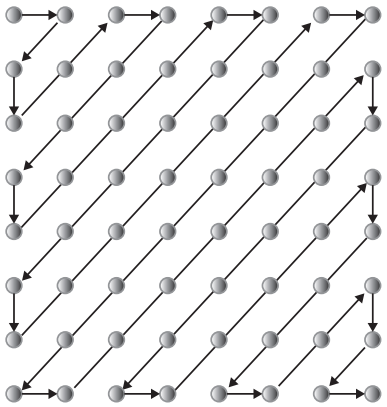


Figure 3.7: Zig-Zag ordering (cf. (122))

ordered creation of the one-dimensional sequence is that the lower frequencies are at the beginning of the sequence and that the later positions of the sequence are zero with a higher probability, which enables a space efficient Huffman encoding [166].

After the preparatory steps are finished, the resulting one-dimensional sequence gets entropy encoded with the Huffman encoder [108]. The goal of this encoding is to save storage by achieving a more optimal representation of the image. The actual encoding is based on the so-called Huffman tables: they usually differ for the DC and AC coefficients as well as for the luminance and chrominance channels, respectively. The JPEG standard provides sample tables that were derived from a large test set of images, however, one could also derive more targeted tables depending on the use-case or a particular type of images (cf. Annex K in [122]). All four Huffman tables used during the encoding are stored analogous to the quantization table inside the JFIF-file, since they are required for the decoding as well. We would like to emphasize here that the described entropy encoding including its preparatory steps is merely a lossless compression technique, i.e. the Huffman encoding itself is *lossless*. For details on the actual algorithm, please refer to [108, 122, 166].

3.3.4 System Details

In the following, we explain in detail how X-pire! works. First, we introduce our mitigation mechanism for the data duplication problem, before we provide in detail the communication protocol of X-pire!. Afterwards, we describe in detail our seamless integration of encrypted data into valid JPEG files.

3.3.4.1 Mitigation of the Data Duplication Problem

Since X-pire! constitutes a pure software solution, we cannot fully solve the risk of illegal copies without posing unrealistic trust assumptions. But our design goal was to mitigate this risk by increasing the costs for an attacker to acquire images or keys on a

large scale and in an automated manner. We decided to base X-pire! on a dedicated X-pire!-Keyserver, since it provides us with great flexibility regarding the expiration dates and allows us to control and protect the access to keys more efficiently. In many settings, distributed approaches are clearly beneficial in terms of performance and scalability. Especially for the setting of X-pire!, we have to admit that a distributed solution would have the advantage that there is not a small number of known keyservers at known addresses that can be queried for keys. However, for X-pire! we believe that the advantages of our dedicated X-pire!-Keyserver in combination with our crawling protection outweigh the ones of decentralized solutions. In the following we do not distinguish between one or several dedicated keyservers at known addresses and list the mechanisms we deploy to protect or limit the access to our dedicated X-pire!-Keyserver:

- we request users to provide us with convincing evidence that they have actually downloaded the resource they want to decrypt in order to increase the costs of retrieving keys on a large scale. This is achieved by storing the hash of the encrypted resource per key and requesting, as well as matching it during the key request process.

- we enforce a rate-limit on key requests both from single IP-addresses and from IP-address ranges.

- we introduce Captchas to prevent the crawling of keys on a large scale. The term Captcha is an acronym for ***C**ompletely **A**utomated **P**ublic **T**uring Test To Tell **C**omputers and **H**umans **A**part* [45, 2] and goes back to Luis von Ahn, Manuel Blum, Nicholas Hopper and John Langford in 2000 [45]. Captchas have been formally introduced by von Ahn et al. in 2003 at EUROCRYPT [3]. The idea of Captchas is to have a test that is easy to pass by humans, but difficult to be solved automatically by computers. In the last ten years Captchas have basically led to an arms race between new artificial intelligence techniques and the hardening of Captcha solutions, since many of them have been broken [51, 244, 243, 215, 42, 130, 154]. Their security is frequently analyzed (for example, [43]) and recently the question is how we can design Captchas that are still easy to solve by humans, but at the same time provide robust security guarantees [44].

  We can request users to solve Captchas at different frequencies and adjust thereby the level of protection for the X-pire!-Publisher and the effort viewers have to invest to actually view images (for example, by requesting one Captcha per image, one image per photo album, etc.). With our dedicated infrastructure for the X-pire!-Keyserver we can easily realize this protection by increasing heavily the costs for an attacker to crawl keys on a large scale.

3.3.4.2 Seamless Integration

The natural choice for the seamless integration of the X-pire!-Viewer and the X-pire!-Publisher into the users' typical workflow was to integrate them into the Web browser. Many modern Web browsers such as Mozilla Firefox or Google Chrome provide APIs to develop feature-rich browser extensions. In certain cases it is even possible to integrate

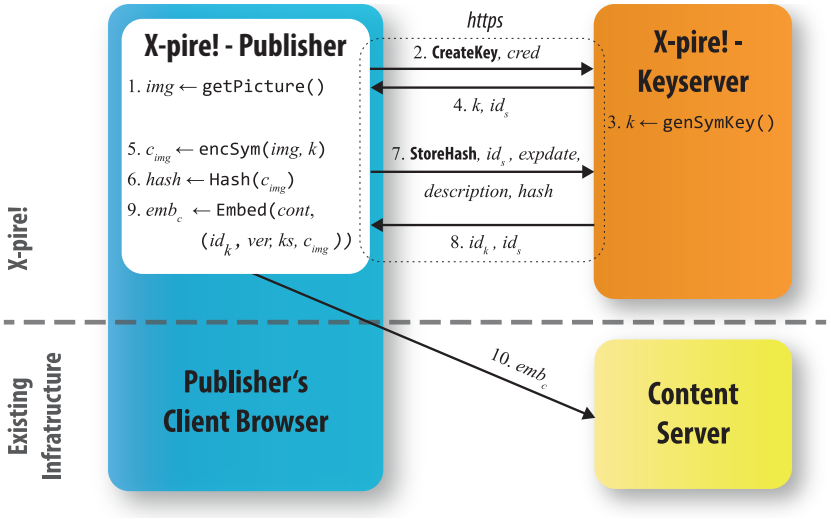


Figure 3.8: Detailed Publication Phase of X-pire!

native libraries to implement high performance features that cannot be implemented directly based on the standard API. Combining a high-level extension with a low-level native library is also the approach X-pire! follows. The automatic replacement of the container images by the decrypted versions of the previously embedded images is based on replacing the link of a resource with a base64-encoded inline version of the decrypted image. For details of the actual implementation, please refer to the implementation of the X-pire!-Viewer in Section 3.3.5.3.

3.3.4.3 X-pire!-Protocol

The X-pire!-protocol is split into the already introduced three different usage phases, i.e., the *Publication Phase*, the *Viewing Phase*, and the *Update Phase*.

Publication Phase The protocol for the publication phase is presented in Figure 3.8. Starting point for the publication phase is an image *img* that the publisher wants to protect for publication (1). In order to protect the image with X-pire!, the X-pire!-Publisher first needs to request a symmetric key for encrypting the image. In order to retrieve the key, the publisher establishes a secure connection (for example, via HTTPs [186]) to the X-pire!-Keyserver and sends a message containing **CreateKey** and *cred* (2). The purpose of the account information *cred* is to authenticate the publisher at the keyserver for a specific account, whereas **CreateKey** triggers the key creation process itself. After the key k has been created (3), the keyserver sends a response to the publisher containing the created key k and a session identifier id_s (4). The publisher uses now the key k to encrypt the image *img*, which yields the ciphertext c_{img} (5). Now, the encrypted image is hashed with a collision resistant hash function (6) and the resulting *hash* is used to link the key k with the encrypted resource c_{img} by sending its hash *hash* along with the **Storehash** command, the previously received session identifier id_s, the desired expiration date *expdate*, and a content description

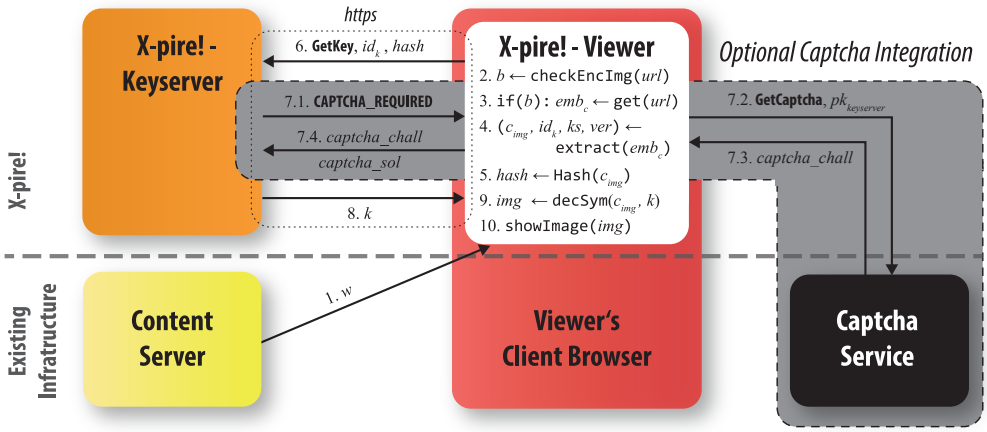


Figure 3.9: Detailed Viewing Phase of X-pire!

back to the X-pire!-Keyserver[7]. The X-pire!-Keyserver confirms the successful process with a final reply containing the key identifier id_k and the session identifier id_s (8). The key identifier is a crucial requirement later during the *Viewing Phase* to identify the required decryption key at the user side.

In order to prepare the ciphertext c_{img} for publication, it finally needs to be embedded into a valid JPEG file: the container image *cont* (9). The embedding includes, besides the ciphertext c_{img}, the key identifier id_k of the key required for decrypting c_{img}, a protocol version ver, as well as the address of the keyserver ks (9). Without embedding c into the container image, it would not be possible to upload the protected image, since c is no longer a valid JPEG file and, therefore, not compatible with existing JPEG upload routines. After the ciphertext c_{img}, ver, ks, and id_k have been embedded into the container JPEG *cont*, we refer to this file as *emb*. *emb* is now ready to be uploaded to the content server (10). We would like to emphasize, that the content server $\mathcal{C}$ is not a collaborating party within the the X-pire!-protocol. In could be an arbitrary JPEG hosting Web server with a JPEG upload routine.

Viewing Phase The *Viewing Phase* describes the protocol of how a user is able to view Web sites that embed X-pire!-protected images. So the images have already been protected during the *Publication Phase* and the user requests a Website w that includes *emb* (1). The X-pire!-Viewer application screens all images embedded to w and checks whether one of them is protected by X-pire! (3). In case such an image is found, the image is downloaded (3) and forwarded to the extraction routine to receive the encrypted image c_{img}, the version of the X-pire! protocol ver, the address of the keyserver ks, and, finally, the identifier of the key required for the decryption id_k (4).

In order to get the key for decrypting c_{img}, the viewer establishes a secure connection to the X-pire!-Keyserver. To receive the required key, the X-pire!-Viewer now first needs

[7]The content description is a high-level description for a particular image or image set that is provided by the publisher. On the one hand, the descriptor is used to assist the publisher in remembering what a particular key was actually used for. On the other hand, the X-pire!-Publisher can use the same descriptor for several images with the impact that they are also encrypted with the same key.

to provide a proof that c_{img} has actually been downloaded. This proof is provided in form of a hash $hash$ of c_{img}, which is computed in step (5). The hash together with the key identifier k_{id} is now send together with the command **GetKey** to the X-pire!-Keyserver. Depending on the settings for the key k at the X-pire!-Keyserver, the keyserver either immediately responds with k, or, if the setting at the keyserver requires the viewer to solve a Captcha, the X-pire!-Keyserver sends back the **CAPTCHA_REQUIRED** (7.1) command. In the latter case, the X-pire!-Viewer sends a Captcha request with the public key of the X-pire!-Keyserver to the Captcha service (7.2), which in turn sends a Captcha challenge $captcha_chall$ (7.3). The viewer solves the challenge and sends it to the X-pire!-Keyserver (7.4), where it is verified in step (7.5) in conjunction with the Captcha service[8]. In case the Captcha is solved successfully, the X-pire!-Keyserver sends the key k back to the viewer. Otherwise, no key k is sent to the viewer.

Update Phase The *Update Phase* allows publishers to configure their accounts at the X-pire!-Keyserver (common account management, for example, to change the password or to delete the account) and to change the settings of keys that have been created in previous *Publication Phases*. For previously created keys one can, for example, change the expiration date (shorten or prolong the expiration date, enforce an instantaneous expiration), change whether Captchas are required, or change the description of keys. The communication between the publisher and the keyserver makes use of a secured connection and the publisher needs to provide the login information *cred* to authenticate for a specific account.

3.3.4.4 Robust JPEG Embedding

In the scenario of X-pire!, where we protect JPEG images on the Internet by adding a digital expiration date, we are required to be compatible with the existing infrastructure for handling JPEG images on the Internet. The existing infrastructure for handling such images mainly consist of static Web site where images are embedded based on the `<img>`-tag of HTML, as well as of Web sites that include upload-routines for JPEG images including post-processing routines as commonly available, for example, in online social networks. Therefore, we are required to embed our encrypted data including our meta information (the address of the keyserver, a key identifier to identify the key required for decryption, the protocol version of X-pire!) for a later key retrieval into a JPEG file that retains valid according to the JPEG standard. The existing handling routines both for rendering images at the client side and for post processing in the upload routines at the server need to keep functioning. To achieve this, we followed two orthogonal approaches of embedding encrypted data in JPEG files:

- The first idea was to embed encrypted data into the header fields of a JPEG file. The embedded data would basically be considered as meta information that does not necessarily affect the processing of the actual image data. This is clearly the method of choice if the publisher controls the publication platform such that re-compression or the like does not occur.

[8]The communication with the Captcha service in step 7.5 is omitted for a better readability of Figure 3.9

- Since the publisher usually does not control the publication platform, we developed the second idea of embedding encrypted data into the actual image data, which was much more challenging. Since the JPEG standard is commonly used to compress images, a potential re-compression of images would most likely destroy our embedded data at least partially. But this would render a decryption impossible: a secure encryption scheme will only allow us to decrypt the ciphertext if we are able to recover 100% of the embedded bits. At the same time, our embedding needs to be fully compliant with the JPEG standard since the resulting cover image needs to retain a valid JPEG to keep compatible with the existing infrastructure.

Embedding into Header Information Embedding the encrypted image as well as the meta information required for a later decryption into the header/meta information of a JFIF-file is not only the fastest method, but also space-wise the most efficient method. In order to embed our data into the JFIF-file, we make use of so-called additional `APP0` marker segments. They are meant for specifying JFIF extensions, which have been available since the JFIF standard version 1.02. The maximum size of such an APP0 segment is limited to 65535 bytes and the actual data per APP0 segment is limited to 65527 bytes: the length field is limited to two bytes and includes in its count all fixed fields (2 bytes length field, 5 bytes as identifier, 1 byte as extension code) except for the APP0 marker itself [99]. But the limited size of a single APP0 segment is not problematic since their number is not limited. So we can add several of these segments in consecutive order. As long as any processing of these files retains the header information, the image itself can even be scaled, re-compressed, etc. However, the upload routines of social networks unfortunately completely strip off existing header information and, thereby, remove our embedded data, which led us to the development of the actual image data embedding.

Embedding into Image Data Since the header information of JPEG images gets stripped off in upload routines of social networks, we now have to embed our encrypted image and the meta information required for a later decryption into the actual image data of JPEGs. Although we do not have to take care about how the container image looks after the embedding itself, this task turned out to be quite challenging. Upload routines of social networks commonly resize and re-compress images according to their own needs; usually they aim at an optimal resolution for the usage in Web sites instead of keeping it at full resolution in order to save storage space.

Countering the resizing of images was actually the straightforward part for achieving the robust embedding into JPEG images: the social networks did only rescale images, if the uploaded image exceeded the height or width limits. Therefore, we decided to scale our container-images upfront to this size[9] to prevent a rescaling, which solves the task.

This left us with the challenge of achieving a robust embedding that survives a potential re-compression during the upload-procedures. The full post-processing of a typical upload routine is presented in the middle box of Figure 3.10 at the example of

[9] for popular Web sites such as Facebook, X-pire! ships with a container image that fits exactly the maximum size

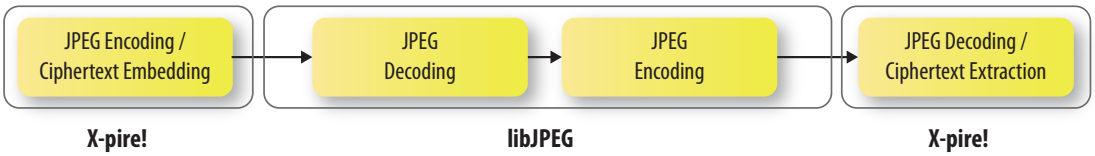


Figure 3.10: X-pire! workflow: Post-processing with libJPEG

the popular libJPEG[10] [121]. In the following we will describe the full development process of the robust embedding including our initial approach that failed to solve the problem.

We started with the idea of anticipating all post-processing steps of social networks in a preparatory step, i.e., to compute a pre-image for the image that is actually published on the Web site after all post-processing steps. Figure 3.11 illustrates this idea. First, the encrypted image and all meta information get embedded into a container image by replacing the actual image data. This process results in emb_c, the container with the embedded data. As long as emb_c is not modified, decryption is possible without any problems. So emb_c is the file we would like to access on the Web site of the social network. To achieve this, we need to compute a pre-image $pre\text{-}emb_c$ of emb_c by inverting all steps of the post-processing process. The decoding sub-process of the post-processing can be ignored, since we can input the pre-image $pre\text{-}emb_c$ without any compression in place such that the decoding process is lossless. In order to invert the decoding process, however, we need to invert every single step starting with the Huffman encoding. However, this turned out to be infeasible since we don't have the corresponding Huffman tables. Without these tables, the inverse decoding step could only be achieved by brute-forcing it.

The next idea was to start one step earlier to prevent the problems with the Huffman-tables, i.e., to prevent the compression applied during the quantization step of the JPEG encoding. This can be achieved very elegantly without the need of inverting every single step of the encoding procedure: First, all required data is embedded into the image data and the image is encoded without compression to a valid JPEG file. Before this image is now uploaded to the social network, the existing quantization table A is replaced by the inverse quantization table of the social network A', which is computed as follows:

[10]The JPEG library libJPEG is a popular open source implementation in C of the JPEG standard with support for various platforms. The library is written and maintained by the Independent JPEG Group [121]. It constitutes the basis for many image manipulation applications such as GD [80] or ImageMagic [111] and is widely used on the Internet. In particular, most of the social networks we are aware of use libJPEG directly, or at least rely on image manipulation applications that were built upon libJPEG.

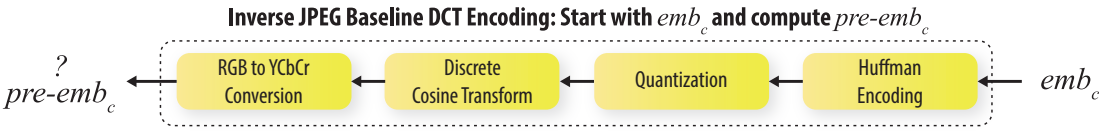


Figure 3.11: Initial idea for the embedding: Compute a preimage for emb_c

$$A = (a_{u,v}) \tag{3.1}$$

$$A' = (a'_{u,v}) = \left(\frac{1}{a_{u,v}}\right) \text{ for } u, v \in \{0, \ldots, 7\} \tag{3.2}$$

Thereby, the initial JPEG decoding of the upload routine applies exactly the inverse quantization step of the later quantization-based compression step during the new JPEG encoding. This solves the problem of achieving a robust embedding that survives the re-compression of upload routines at least for our mathematically correct implementation according to the JPEG standard. However, when using the popular JPEG library libJPEG, the situation changes: Our approach breaks down completely. When using libJPEG our reconstruction rate dropped down to only 10%, which is far too low for meaningful improvements or corrections based on error-correcting codes. Since libJPEG is used within the upload-routines of all social networks X-pire! is supposed to support, this problem rendered also our second approach infeasible. Without reliably achieving a reconstruction rate of 100%, a later decryption of protected images is impossible.

We analyzed libJPEG to understand why our own JPEG implementation performed that differently and it turned out that libJPEG is highly optimized for an improved performance. In several cases, performance improvements are favored in contrast to precision, for example, by introducing rounding errors. Based on these insights, we developed our final approach for robustly embedding encrypted data into JPEG images. The new approach is based on libJPEG and ensures that embedded data can be fully recovered even after a libJPEG-based re-compression during the upload to a social network, but its integration into libJPEG was rather challenging. The workflow of libJPEG does not comply with our requirements for X-pire!. The library is implemented in C and it is commonly used for reading a source image, decoding the image, and, subsequently encoding it again with different settings. It does not provide an API for accessing single steps of the encoding/decoding process (for example, only the discrete cosine transform or only the quantization, cf. Figure 3.12), since single steps are highly intertwined to improve the performance of the library. The library combines, for example, the image scaling procedure with the DCT computation. For X-pire!, however, we have to closely interact with the original encoding and decoding routines to integrate our robust embedding and this required an in-depth understanding of the original libJPEG structure and functions.

In the following, we describe in detail how the final embedding approach of X-pire! works and how it was integrated to libJPEG. Since the quantization tables used by social networks are publicly known[11] and not subject to frequent changes, the final embedding focuses on an embedding that survives this particular compression step. To achieve this, data is already embedded right before the level shift and the implied conversion to a signed integer representation before the discrete cosine transform (DCT) is applied (cf. Figure 3.12). The DCT implementation receives as input the 8 by 8 blocks and each value inside the blocks constitutes an 8-bit unsigned Integer value which is subsequently converted into signed Integer values[12]. We embed now before the level shift in each

[11] They are part of every published JPEG image

[12] The level shift is also applied inside the DCT routine.

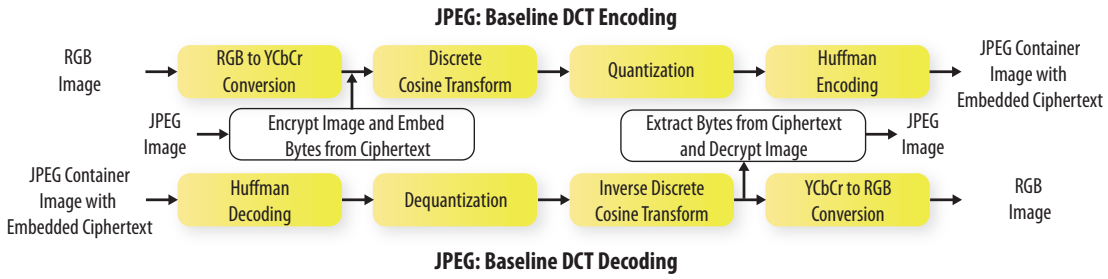


Figure 3.12: X-pire! integration into the JPEG Encoding and Decoding routines of the Baseline DCT mode of operation

of the single values of each block our data into the most significant bits (MSBs). By default, we only embed data into the two most significant bits. The intuition behind this is that the most significant bits have the highest impact on the actual image and are, thereby, the last bits affected by a potential compression. The number of bits unaffected by the later compression based on the quantization step depends on the compression rate and, thus, on the values in the quantization table. We provide the quantization table of Facebook at the time of the X-pire! development in Figure 3.6. Depending on the quantization table it might actually be possible to embed data into more than the first two bits, but it is also possible that only less data can be embedded. As future work we consider, therefore, to implement an adaptive approach that – depending on the actual values inside the quantization table – automatically varies the amount of bits used for the embedding. Our analysis showed that our results can be further amplified by setting always the bit next to the bits used for embedding data to one (for example, bit 5 when using bits 6 and 7 for the embedding.). This prevents the bits used for the embedding from being affected by rounding errors, since even major deviations in the lower bits will not reach the upper bits.

Although the described approach of robustly embedding data produces for libJPEG far better results than our previous designs, some data loss still occurs. However, this loss can be handled. During our analysis of the new embedding (the analysis included the upload and download of images from social networks), we never encountered more than 5% loss. In order to cope with this amount of loss, we decided to introduce a state-of-the-art error correction mechanism. In principal any recent error correction mechanism would work and we decided for X-pire! to make use of so-called Reed-Solomon codes [183] in a configuration (N, K) of $(255, 191)$. This configuration allows the successful recovery of 191 bytes of actual data in case the loss of data is at most about 12,5% [P8]. To correct the errors in 191 bytes, 64 bytes of information are included for the error correction, which sums up to 255 bytes per error correction unit.

Based on the described embedding, the Reed-Solomon codes for error correction, and our embedding into the two most significant bits, X-pire! can now embed a maximum of 3/16 of the original payload into the luminance channel for the target platforms, i.e., the mentioned social networks such as Facebook. In principle, we can embed 2 bits per byte, which provides us with 1/4 of the original payload of the luminance channel and this is further reduced by 25% for the error correction symbols. The same holds true for the chrominance channels, but since these channels are frequently subject to sub-sampling,

they are currently not used by X-pire!. The total amount of data that can be embedded using the described method is fully sufficient for uploading encrypted images inside of the container images to social networks, especially since X-pire! re-compresses images upfront with the settings of the particular social network[13]. Thus the approach provides us with the necessary basis to fulfill our goal of implementing the digital expiration date for social networks.

3.3.5 Implementation

In the following, we describe the implementation of X-pire! as it was carried out by Stefan Lorenz (client side) and Julian Backes (server side) for the production version of X-pire! [P1]. The client-side implementation of X-pire! consists of a browser extension for Mozilla Firefox version 3.5 that interfaces a native library. The native library was developed for X-pire! and includes all functionality required for the *Publication Phase* and the *Viewing Phase*. It can be interfaced based on the Gecko/XULRunner SDK 1.9.1 for developing so-called XPCOM components, which can provide an interface to native libraries. At the server-side, X-pire! provides the X-pire!-Keyserver including the Web interface for key management, which mainly relies on state-of-the-art Web server technologies.

3.3.5.1 The X-pire!-Library

The Firefox browser extension at the client side contains the X-pire!-Publisher and the X-pire!-Viewer. The publisher provides all functionality required to protect images with an expiration date for a later publication, whereas the X-pire!-Viewer provides the functionality required for decrypting and viewing already published images. Both the publisher and the viewer share a native library written in C and C++ that provides the full X-pire! toolchain for the actual image handling. Relying on such a native library for the image manipulation tasks allows us to easily integrate existing and well-tested third-party libraries for performance critical tasks: This includes the symmetric encryption and decryption of images based on the OpenSSL library, hashing based on the OpenSSL library, our embedding routines in conjunction with libJPEG, as well as a custom implementation of Reed-Solomon codes for the error correction[14]. In particular, X-pire! uses the advanced encryption standard (AES) [155] with cipher block chaining (CBC) [147] mode and a key size of 256 bits for the symmetric encryption and SHA256 [119] as hash function for both the *Publication* and the *Viewing Phase*. The embedding as described in Section 3.3.4.4 is integrated into the libJPEG version 8b by a custom module that is called inside the function for the discrete cosine transform (for embedding data), or inside the function for the inverse discrete cosine transform (for the extraction of data), respectively. Both the header-based embedding routine (cf. Section 3.3.4.4) and the libJPEG-based embedding routine (cf. Section 3.3.4.4) are finally integrated into a custom C++ library with a well-defined API for using both approaches. Since the browser extension is required to call functions of our

[13] It would not make sense to provide images with better quality, than it would be done by the social network itself

[14] The library for Reed-Solomon codes was implemented by Stefan Lorenz.

library directly, it was necessary to implement an XPCOM component and to provide a wrapper for our library[15] for the communication between the high-level extension code and the native library. The XPCOM framework allows us to call native functions in the same way as JavaScript or XUL functions are called by the extension. The whole network communication of the extension is not integrated into the library, but uses the `XMLHTTPRequest` function in JavaScript for interfacing the X-pire!-Keyserver via HTTPS.

3.3.5.2 The X-pire!-Publisher

The publisher is part of the browser extension and provides users with the interface for protecting images with an expiration date. The menu is shown on the right side of the browser window (cf. Figure 3.13, the window can be hidden). Depending on the chosen target (own Web site or a social network such as Facebook), either the header-based embedding method or the method for embedding data into the actual image data of the container image is used. Below the target selection, users are provided with a file selection pane where even multiple files can be chosen[16]. At the bottom of the publisher menu, users can configure the desired expiration date. The advanced pane provides the interface for configuring the X-pire! account for the X-pire!-Keyserver.

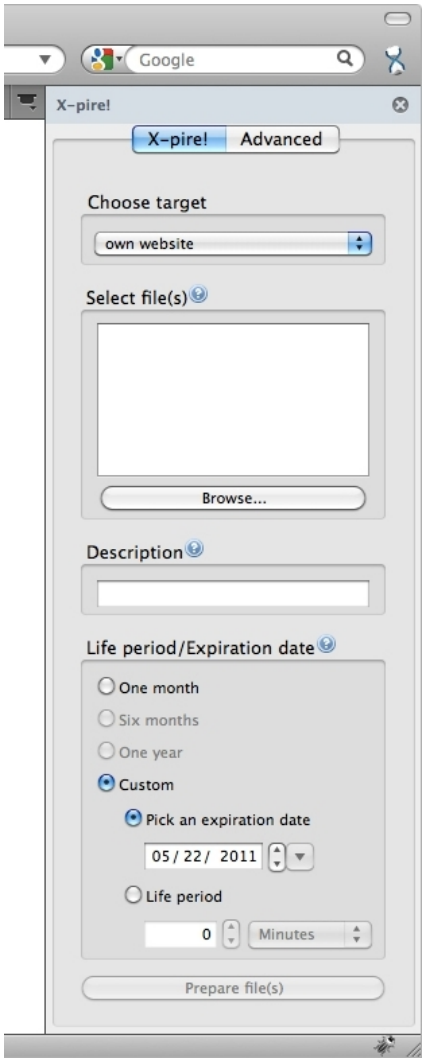


Figure 3.13: X-pire!-Publisher: User-interface (Screenshot by Julian Backes)

3.3.5.3 The X-pire!-Viewer

The viewer does not provide a special user interface the user needs to interact with, it is either enabled or disabled. The only interface required by users of the X-pire!-Viewer is the one for configuring a white-list of Web sites where the X-pire!-Viewer should be enabled. The extension is automatically disabled for all other sites to improve the performance (cf. Section 3.3.6 for details on the performance impact of the check for X-pire!-container images)[17].

[15] All Firefox versions supported by X-pire! do not support direct calls without an XPCOM component.
[16] As described in Section 3.3.4.3, multiple files with the same description may share the same key.
[17] The current version of X-pire! ships with a white-list for social networks.

The core functionality of the X-pire!-Viewer is fully transparent to the users. The handler functions of the extension check in the background whether a <img>-tag of a white-listed Web site embeds a X-pire!-protected images. If such a container image is found, the extension immediately extracts the encrypted image including the meta information. Afterwards it contacts the X-pire!-Keyserver at the address specified in the meta data and requests the key required for decrypting the embedded image from this X-pire!-Keyserver. Once the key is received and decryption succeeded, the extension modifies the document object model (DOM) tree of the Web site with the embedded X-pire!-image that is currently loaded and replaces the <img>-tag referring to the X-pire!-container image by a new <img>-tag containing the full, decrypted images in *base64*-encoding. Although RFC 2397 on *The "data" URL scheme*, which defines this usage of *base64*-encoded inlined data as URL, constitutes only a proposal for a standard[18] [138], it is today widely implemented: all major browsers support it[19].

3.3.5.4 The X-pire!-Keyserver

The keyserver is implemented using Scala, a functional and object-oriented programming language that runs in the Java Virtual Machine and allows a direct interaction with Java code [218]. In addition, X-pire! leverages for the X-pire!-Keyserver the Lift framework [131] for developing its Web application including the key management interface and PostgreSQL [175] as the database back-end for storing the actual keys. The keyserver integrates the Google reCAPTCHA service[20] [88] to prevent large-scale crawls of keys (cf. Section 3.3.4.3 for the detailed protocol). The requirement to solve such Captchas is supposed to heavily increase the workload for potential attackers. The activation of this service can be chosen by the publisher either during the initial key request or via the key management interface. Moreover, the communication interface of the X-pire!-Keyserver enforces the usage of HTTPS to ensure data confidentiality, the same holds true for the whole communication with the reCAPTCHA service.

3.3.6 Evaluation of X-pire!

In the following, we provide the results of our X-pire! evaluation based on [P1].First, we provide the results of the performance evaluation for both the X-pire!-Viewer and the X-pire!-Publisher. Additionally, we compare the quality of the original images in social networks with the quality of our embedded images and explain how much data can be embedded for which network[21]. Most important was the evaluation of the client-side application (the Firefox Browser extension), since its impact on the actual browsing performance will have a major impact on the users' acceptance and, thereby, on the deployability of X-pire!. In addition, we analyzed the server-side implementation regarding its scalability.

[18] The Internet Engineering Task Force published the proposal by L. Masinter already in 1998 [138]

[19] Verified on May 18th, 2014 with Apple Safari 7.0.3, Mozilla Firefox 29.0.1, Google Chrome 34.0.1847.137 m, and Microsoft Internet Explorer 11.0.9600.17107

[20] According to Google, reCAPTCHA constitutes the "most widely used Captcha provider in the world" and their Captchas are used to digitize text, annotate images or build machine learning datasets.

[21] The numbers and quality settings are based on the development state of the online social networks when we designed X-pire!

3.3.6.1 Client-side Performance

The evaluation of the client-side performance was conducted on a commodity notebook with an Intel Core 2 Duo CPU (2.2 GHz, 2cores, 2 threads) and 4GB RAM running Mac OS X 10.6.4 and our measurements focused on three different aspects of the X-pire! extension: the performance impact of the browser extension with the X-pire!-Viewer when visiting Web sites, the performance of the embedding itself, and, finally, the full viewing process of X-pire! protected images (image extraction, image decryption, and image rendering).

First of all, we measured the performance overhead introduced by the Firefox browser extension when browsing normal Web sites. In order to achieve great usability and a widespread acceptance, it is essential that users do not recognize any negative performance impact during their day-to-day tasks when using the browser. The impact on these day-to-day tasks was measured by requesting custom static HTML-pages[22] with a different number of embedded images (no X-pire!-protected images) 50 times per series while the X-pire! extension was activated. The results are compared to the baseline experiment with the same settings but without having the X-pire! extension activated. As shown in Figure 3.14, there is no measurable performance impact when viewing Web sites without X-pire!-protected images.

In our next experiment, we measured the performance of encrypting and embedding an image into an X-pire! container image, both with the header-based embedding and the embedding into the image-data. The experiment was executed again in series of 50 runs per number of embedded images. The results are provided in Figure 3.15. As expected, the header-based embedding clearly outperforms the embedding of the encrypted image into the image data and both approaches scale linearly. The major time consumption for the header-based embedding is related to the encryption, which is similarly present for the image data-based approach. The embedding into the header does not consume much time since it is roughly the time for writing the encrypted data to a file. In other words, the difference in time between both approaches is roughly the time that is required for the embedding into the image data.

Finally, we analyzed the time it takes to process Web sites that may embed X-pire!-protected images. We measure for both embedding approaches the time it takes to detect whether an image is an X-pire!-protected image and compare the results to the full process of extracting and decrypting such a protected image. This measurement relies as the two previous benchmarks on the setting of 50 runs per series and each series containing a Web page with a different number of embedded images. The detailed results are provided in Figure 3.16. The header-based embedding outperforms also for the extraction and decryption process the image data-based embedding method. In particular, as shown in Figure 3.16, already the check for a protected image using the image data-based approach takes as long as the full extraction and decryption process of the header-based embedding approach. Our experiments further revealed that neither of the two approaches scales linearly. We expect that this was due to the implementation

[22] It is difficult to measure the performance overhead while viewing existing Web sites such as social networks since they embed too many external resources and their linked content (for example, online advertisements) changes frequently. For many linked resources changes occur even every time a page is loaded.

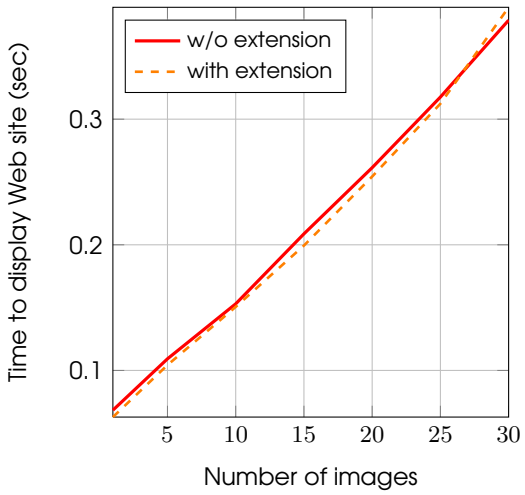


Figure 3.14: Performance impact of the Firefox browser extension (P1)

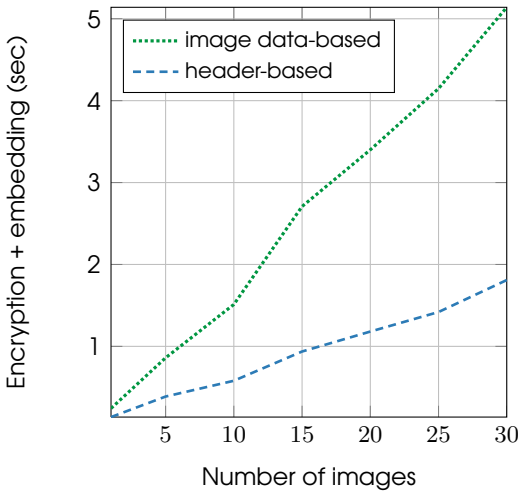


Figure 3.15: Time to create X-pire!-protected images (P1)

of the JavaScript DOM handlers and events in the Mozilla Firefox browser [P1, 150].

3.3.6.2 Server-side Performance

Besides the X-pire! browser extension at the client side, we also evaluated the performance of one central keyserver regarding its scalability. The evaluation was performed on a commodity desktop computer with an Intel Core i3 540 CPU (3,06 GHz, 2 cores, 4 threads) and 4GB RAM running Ubuntu 10.04 Server Edition. On top of the Linux operating system we used the Java-based Web server Jetty 6 [116] and assigned 2GB heap space to it. The benchmarks itself were performed with ApacheBench [1]. Based on this experimental setup, we were able to create about 820,000 sessions until the 2GB memory limit for Jetty was reached, which allows already a pretty reasonable amount of users per server. However, this constitutes only a rough upper bound since Jetty cannot display the number of active settings [P1]. After the 2GB memory limit was reached, the garbage collector of the Java virtual machine slowed down the system until it fully stopped. Besides the analysis of the maximal amount of parallel sessions for this configuration, we analyzed the scalability of the X-pire!-Keyserver regarding the maximum CPU load. During our measurements we were able to execute roughly 3000 requests per second within one single session. This amount of requests pushed the CPU load to the limit on all cores with a CPU usage between 90% and 100% according to htop [P1, 107].

3.3.6.3 The Embedding

Although most of the upload routines for social networks rely on libJPEG, they apply different settings for the re-encoding of images, which has influence on the amount of data X-pire! can embed. Table 3.2 summarizes the amount of data that can be

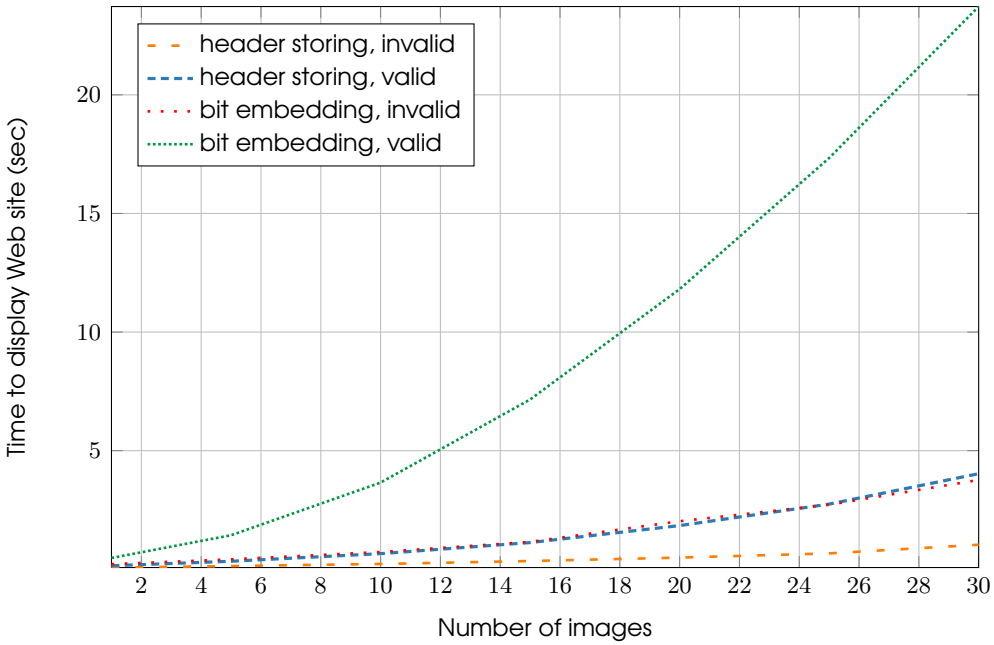


Figure 3.16: Performance of the X-pire!-Viewer for both embedding approaches (P1)

embedded to *Facebook*, *Flickr*,and *wer-kennt-wen*, respectively; the max. resolution provided in Table 3.2 refers to the max. resolution of images that can be uploaded without getting rescaled[23]. Overall, the amount of data that can embedded suffices to store images at a reasonable quality, especially when compared to the image quality used by online social networks. In order to compare the quality of X-pire!-images with original images from social networks, we provide in Figure 3.17 a X-pire!-protected image that was downloaded from Facebook and in Figure 3.18 the same image, but now downloaded from Facebook without any X-pire! processing.

3.3.6.4 Security Evaluation

In the following, we discuss the security of X-pire!. A core assumption of X-pire! is that once a particular expiration date has passed by, the X-pire!-Keyserver deletes all keys required for decrypting files with this particular expiration date. If an attacker was able to retrieve keys or even decrypted content after the expiration date, this would break the system for a particular file. Since an attacker is in principle able to take the role of an average user, the attacker can also retrieve keys and decrypted content as long as the expiration date has not been reached. So if an attacker knows already before the expiration date what files to compromise, a targeted attack is possible. However, we argue that this is not common for our use case. When people publish personal pictures on social networks: who knows already years upfront, which data is required to be detrimental for the future lives and career of this people? Since this is unclear, attackers

[23]The values are based on the social networks' configurations that were present during the design and evaluation phase of X-pire! in 2010/2011.

would have to store keys or decrypted images on a large scale, which is a huge effort. Due to our Captcha-based crawling protection and the rate limits, only a large fraction of collaborative attackers investing huge efforts in terms of man power and resources would be able to remove the X-pire!-based protection on a large scale. We don't expect this, however, X-pire! does not protect against this as well. In particular, X-pire! was *not* designed to achieve this.

3.3.7 Discussion and Limitations

X-pire! has raised a huge discussion in Germany about whether the Internet should be capable to forget and about how X-pire! contributes to this potential goal. From our point of view, the Internet clearly needs better mechanisms to protect the users' privacy and, in particular, the privacy of minors – even if they intentionally publish personal data. It should be possible to remove data that is published by children or teenagers and potentially detrimental for their future life and careers. Ideally, published data should completely stay in the controls of minors as well as in their parent's control. This is also in line with the draft for the new *European General Data Protection Regulation*. The regulation includes an explicit article that mandates the "right to be forgotten and to erasure" (Article 17 in [69]).

Although X-pire! received an overwhelming attention in the German media – reports on X-pire! occurred in most of the major printed press, in major TV channels, as well as in the radio –, X-pire! was also heavily criticized especially in online media for not providing strong security guarantees. Most of the criticism came up because people had wrong expectations and knowledge about the functionality of X-pire!. Our system was designed as a software solution based on the infrastructure that existed in 2010 and our design is built around the assumption that users of the system are benign. Admittedly, benign users can copy the content (for example, images) as well as the keys required for decryption until the expiration date has been reached. But since we assume that attackers do not know in advance the data they might need after it has expired, it would require a large collaborative action to break the full system and to remove all expiration dates. This is not possible according to our assumptions. We are convinced that X-pire! is the technical limit for a solution purely in software without making unrealistic trust assumptions on the underlying system. If one considers malicious users for a software based solution, one could only move the border where it fails by making trust assumptions, for example, on the operating system. But depending on the user rights (for example, root), no effective prevention can be assumed for software solutions. Even if these rights are not granted by default or when we consider systems where people usually cannot gain root access, system breaks are quite popular (for example, rooting on Android or Jailbreaks for iOS). In order to prevent wrong expectations, we list in the following both the functionality X-pire! was designed for and the limitations of our system. In particular, X-pire! was designed to provide the following functionality:

- The encryption of JPEG images and associating them with an expiration date.
- Uploading these images to social networks such as Facebook or Flickr.

Social Network	max. Resolution	Image Bytes	Bytes incl. ECC	Bytes w/o ECC
Facebook	720x720px	468000	117000	87750
Flickr	1024x1024px	976896	244224	183168
wer-kennt-wen	620x620px	341000	85250	63937.5

Table 3.2: Amount of bytes that can be embedded into the container image (P1)

- Viewing the images after the upload to a social network with our browser plug-in in supported browsers.
- Integration of Captchas to heavily increase the costs for an attacker to collect large amounts of keys or decrypted images.

The design of X-pire! does **not** provide protection against:

- Users that intentionally copy images after they got decrypted; this is always possible for solutions purely in software without posing unrealistic trust assumptions.
- Users installing malware to collaboratively collect and store keys on third-party servers whenever a picture is viewed (a proof of concept tool called *Streusand* was presented by Federrath et al. [72]); this is comparable to intentionally copying images that could also be stored unencrypted on another server.

Another point of criticism was related to the choice of a dedicated keyserver for X-pire!. As a matter of fact, relying on dedicated keyservers provides attackers with a clear target. They simply know where the keys required for decryption are stored. However, a dedicated keyserver infrastructure also provides several advantages: First of all, it eliminates the limitations on the expiration dates dictated by the infrastructure that several related systems have to cope with [82, 46, 184]. Second, publishers that do not want to trust any keyserver infrastructure get to chance to set up their own infrastructure. Finally, a controlled dedicated keyserver infrastructure allows us to deploy access control restrictions to prevent large-scale key retrievals. Although we did not implement other concepts for the keyserver, we would like to emphasize that the infrastructures of related approaches such as Vanish (DHTs) [82] or EphPub (DNS caches) [46] could also be used by X-pire!. One could also think of reducing the risks of a single broken or malicious keyserver by using a k out of n threshold-based encryption scheme [202] and storing individual key shares on several different dedicated keyservers.

3.4 X-pire 2.0

The motivation for designing X-pire 2.0 was to solve the central problem of X-pire!: the data duplication problem (DDP). An attacker should never be able to copy content or the keys required for decryption. X-pire 2.0 was designed as a flexible system for publishing data with an expiration date that solves the data duplication problem and provides robust security guarantees. It leverages the robust embedding of encrypted data into JPEG images of X-pire! and extends it with a robust solution to the data

Figure 3.17: Image uploaded using X-pire!; it has the identical image quality as the Facebook image on the right side

Figure 3.18: Image uploaded using only Facebook

duplication problem based on secure hardware. Thereby, X-pire 2.0 constitutes the first system for a digital expiration date for digital data that achieves a robust protection against the data duplication problem and provides mechanisms for a seamless integration into the existing infrastructure at the same time.

The workflow of X-pire 2.0 is similar to X-pire!. After the publication of protected data it is possible to modify the expiration date of published data, or even to enforce an instantaneous expiration. The latter is achieved by either changing a previously defined expiration date or by adding an expiration date at all. Once the expiration date set by the publisher is reached, published data becomes unavailable. Since X-pire 2.0 builds up on the same embedding techniques as X-pire!, it targets the same scenarios: 1) publisher have control over the publication platform, for example, their own Web server, and 2) publication platforms like modern online social networks. For the latter, we show for images how our approach allows a lightweight integration into the existing infrastructure: in particular, X-pire 2.0 can be used to publish protected images via Google+ and Flickr. The security of the system is based on the fact that after the publisher protected the data to be published, it will never be transmitted or processed in *cleartext* by any untrusted entity. Please note that we do not consider consumers of protected content as trustworthy entities, as an attacker could easily take the role of a legitimate user (for a detailed overview of our assumptions, cf. Section 3.4.2.3 and 3.4.2.4).

3.4.1 Contributions

In more detail, X-pire 2.0 offers the following major improvements in contrast to X-pire!:

1. X-pire 2.0 allows publishers to fully stay in control of their published data by providing robust guarantees against attackers. X-pire 2.0 does no longer assume a trustworthy and benign user of the system.

2. X-pire 2.0 provides the first solution to the DDP. This is achieved by basing X-pire 2.0 on ARM's trusted computing framework *TrustZone* [4, 13]. This ensures that neither legitimate users nor attackers can access and copy content before the expiration date is reached[24]. (After the content expired, creating such copies is of course impossible by design.) ARM TrustZone is available for a series of ARM processors such as the Cortex-A9 [14]. These processors constitute a large share of ARM's 90% market share for smartphone processors [221].

3. X-pire 2.0 provides an effective profiling protection against the X-pire 2.0-Keyserver by leveraging private information retrieval (PIR) techniques based on Oblivious RAM and secure hardware. This profiling protection counters the potential for an efficient profiling of key requests by users through the central keyserver. The central keyserver is required by our solution for the DDP.

The overall system provides a general publication framework for digital content and could easily be adapted for data types other than images, as well as for other publication platforms than social networks. Especially in cases where the publisher maintains the publication platform, our approach is straightforward to adapt. We have implemented our system for the Google Android platform[25] and conducted performance measurements to demonstrate its efficiency.

3.4.2 System Overview

X-pire 2.0 provides a digital expiration date for digital data and achieves robust security guarantees. In the following we give a high-level introduction to the X-pire 2.0 protocol and discuss the technical challenges during the design phase. Moreover, we discuss the systems' underlying prerequisites and assumptions, and outline the attacker model for the system.

3.4.2.1 High-level view on the protocol

The usage of X-pire 2.0 is split similar to X-pire! in three different phases (cf. Figure 3.19), the *Publication Phase*, the *Viewing Phase*, and the *Update Phase*.

Publication Phase: The publisher $\mathcal{P}$ encrypts data with a symmetric key and stores the key together with an expiration date on a dedicated keyserver $\mathcal{K}$. From then on, eligible devices can access this key until the corresponding expiration date set by the publisher is reached (cf. Section 3.4.4.1). After encrypting the data, the publisher $\mathcal{P}$ embeds the encrypted data into a container image that is compliant with the desired content server $\mathcal{C}$ and publishes the container on this content server $\mathcal{C}$, for example, on a public Web server or a social network.

[24] Clearly, taking photographs of the monitor with an external camera is always possible.

[25] The current limitation constitutes a prototype purely in software. Due to the missing availability of openly programmable and documented TrustZone hardware, the system is currently providing no security guarantees.

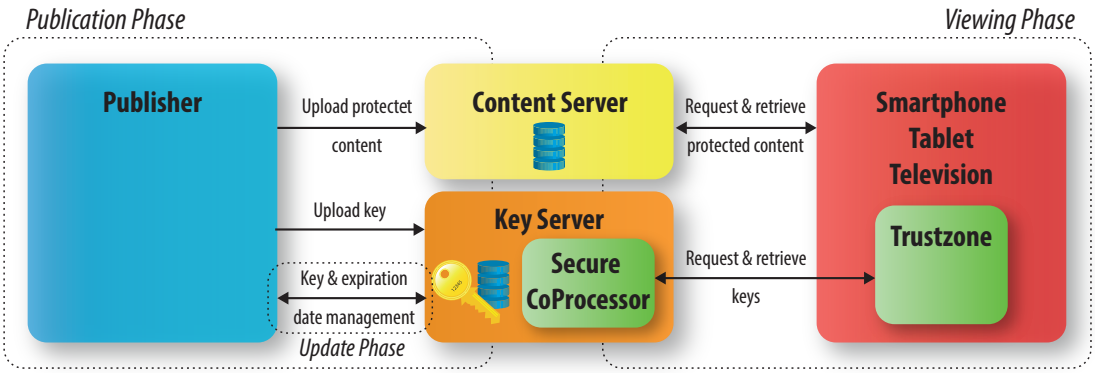


Figure 3.19: X-pire 2.0: A high-level overview

Viewing Phase: In order to view protected content, the browser forwards it to the TrustZone to initiate the corresponding key request. After the key is retrieved from the keyserver, the key is used to decrypt the protected content and to subsequently display it to the user from inside the TrustZone.

Update Phase: The keyserver supports key management functionality, which can be used during the *Update Phase* to modify the settings for already created keys. In particular, the management functionality allows publishers $\mathcal{P}$ to prolong and shorten the expiration date of keys. The latter can even be used to let data expire instantaneously. Once the expiration date set by the publisher $\mathcal{P}$ for a particular key k is reached, published data encrypted by k becomes unavailable.

3.4.2.2 Technical challenges

In order to achieve our goal of implementing a comprehensive expiration date for digital content with robust security guarantees, we needed to solve several technical challenges:

Solving the data duplication problem. First and foremost, we needed to solve the data duplication problem. Public responses to previous deployment attempts of a digital expiration data have shown that the DDP constitutes the main reason why these approaches did not find a widespread deployment and acceptance in practice. In a nutshell, all existing approaches can only provide robust security guarantees if all users that can access a person's data before its expiration are fully benign. Although data is always stored encrypted, both the key and the content are available in the clear when a legitimate user reads the protected data before the expiration date is reached. Without further protection, an attacker could simply store and republish keys or content and, thereby, fully remove the existing protection. In order to solve the problem, it is of central importance that an attacker, even if taking the role of a benign user, is neither able to copy content after decryption, nor able to copy the decryption key at any point of time. Without imposing unrealistic trust assumptions, the only way to achieve this is by preventing users from accessing both the keys and the content. And this is exactly where the difficulty lies: users need to view content without getting digital access to

the decryption keys or to the content that has been decrypted for viewing. We would like to stress that already a screenshot is a digital copy that needs to be prevented by any means. However, even if screenshots can be prevented, how is it possible to prevent somebody from capturing the video signal to the screen?

X-pire 2.0 assumes that an attacker can both take the role of a legitimate user and manipulate the off-the-shelf OS of a viewing device. Therefore, we move all processing of encrypted content out of the normal operating system and use the ARM TrustZone framework, a trusted computing approach integrated in many embedded devices based on ARM processors [4, 13], such as today's smartphones, tablet computers, or modern flat screen TVs (for details on the TrustZone, cf. Section 3.4.4.2). When a user wants to view protected content, it is forwarded to a trustworthy environment (the so-called *secure world*); it is the counterpart to the *normal world* with the normal untrusted OS) that is unreachable for attackers and even honest users or the normal world operating system (OS). The key request for the decryption key is only executed by the trusted OS running inside the secure world, which is also the only place, besides the keyserver, where the decryption key can be stored (at most, until the expiration date). Decrypting content and subsequently displaying it is also handled inside the trusted environment. Protected content is only shown on displays embedded in the supported devices and not on external screens. Furthermore, users are not able to extract content or keys from the trusted environment. In case keys are transmitted over the Internet, they are only delivered via secure channels after a mutual authentication. In order to ensure that only devices incorporating an eligible TrustZone are able to access keys, we decided to make use of a central keyserver. Based on the PKI used, the keyserver can easily identify whether a key request was issued by an eligible device before the decryption key is sent in return. If keys were stored or derived in a distributed way on the Internet, as, for example, in [82, 184], this server-side authentication enforcement could not be achieved.

Protection against profiling content viewers. In order to achieve our solution for the DDP, we need to rely on a dedicated centralized keyserver infrastructure for storing the content decryption keys. Using a centralized approach for the keyserver introduces the potential for an efficient profiling of users by tracking decryption key requests, as the keyserver potentially learns the content users have viewed. However, this problem does not only exist for centralized approaches. In the decentralized setting a similar profiling could be achieved by colluding attackers. To a certain extent, users (especially the publisher) have to trust the keyserver anyway since it is central to X-pire 2.0's security that keys are not leaked in any way. Nevertheless, many users want to prevent any possible profiling and do not agree to provide the keyserver with a detailed profile of requested keys. We counter the risk of users being profiled by using PIR techniques based on Oblivious RAM and secure hardware [236]. Thereby, the keyserver is unable to learn details about key requests and resulting responses.

Compliance with the existing infrastructure Since we follow a similar paradigm as existing approaches and protect content by encrypting it, the resulting files are usually incompatible with existing infrastructure. But most personal information that strives for protection (for example, private and potentially unflattering images) is nowadays

published on existing infrastructure like social networks, where control over supported file formats is out of reach for publishers. Therefore, it is necessary to embed the encryption such that the original file format is retained. We leverage the embedding developed for X-pire! (cf. Section 3.3.4.4) and show for the use-case of JPEG images how this can be achieved for social networks such as Google+ and Flickr.

Seamless, one-click integration. We have implemented our system for the Google Android platform[26] and conducted performance measurements to demonstrate its efficiency. The software integrates seamlessly into the user's workflow when surfing on the Internet. We implemented the X-pire 2.0-Publisher and the X-pire 2.0-Viewer as two distinct Android apps. To publish an image, users can either use the X-pire 2.0-Publisher to directly take a photo (using, for example, an Android-based smartphone or camera), or the publisher can load an existing image into the app. Afterwards, the desired expiration date is entered and the image gets protected. Subsequently, the user uploads the protected content to the desired publication platform. The X-pire 2.0-Publisher allows both to take photos and to load images within the app[27], before they get protected right afterwards. In the viewing process, the user surfs the Web using the X-pire 2.0-Viewer app: in case protected content is detected, X-pire 2.0 processes the content and displays it.

3.4.2.3 Requirements and Assumptions

In order to fulfill the goals of X-pire 2.0, we rely on trusted hardware to protect *cleartext* versions of published content and the corresponding decryption keys. This provides the technical basis to solve the DDP. Furthermore, we rely on secure hardware to protect the privacy of users (viewers of content) against the keyserver. In particular, X-pire 2.0 has the following requirements:

- The keyserver needs secure hardware for the implementation of private information retrieval based on Oblivious RAM in order to prevent a possible profiling of user-based key requests by the keyserver.

- The device used for viewing protected content needs to be based on an ARM *System-on-a-Chip* (SoC) incorporating the TrustZone, an extension for trusted computing introduced by ARM (cf. Section 3.4.4.2 for particular TrustZone features required by X-pire 2.0). Typical devices based on ARM SoCs include smartphones, tablet computers, and recently also TVs. In the future, however, the concept could also be adapted for other trusted computing platforms, as long as all required functionality is provided. One could, for example, follow the spirit of Flicker, a framework providing secure code execution on off-the-shelf platforms with a minimal trusted computing base [143].

[26] Since openly programmable TrustZone hardware is not publicly available, our implementation is purely in software and does not provide the desired security guarantees.

[27] Most modern smartphones and tablet computers incorporate a camera. Additionally, there are even normal digital cameras running the Android OS

- The device integrating the TrustZone needs an embedded display so that the signal between graphics card and display cannot be easily intercepted and recorded (for example, smartphones, tablet computers, or smart TVs.).
- We need a secure hardware clock inside TrustZone to correctly handle expiration dates when keys are cached.
- We need a standard public key infrastructure (cf. Figure 3.21 for the involved entities).
- The code base running inside TrustZone needs to be error-free and protects both content and keys (for example, the decryption keys). Keys are never leaked out of the *secure world* inside the TrustZone, since the *secure world* will only execute trustworthy and endorsed code. An alternative solution would be to ensure that the trusted code base inside the *secure world* ensures a sequential and non-interfering execution of different applications to prevent them from accessing each other's sensitive data.

3.4.2.4 Attacker/Threat model

Previous approaches that implement an expiration date for digital content (for example, [82, 46]) assumed a benign user and had, if at all, only weak protection against an attacker that takes the role of a legitimate user. In contrast to these approaches, we provide a solution to the DDP and assume that the attacker can take the role of a benign user. In particular, we assume that the attacker is

- capable to take the role of a legitimate user of X-pire 2.0.
- able to manipulate the normal world OS on the devices on which he is executing the X-pire 2.0-Viewer and X-pire 2.0-Publisher.
- able to intercept, replace, and modify network traffic.
- **not** capable to conduct a lab attack against the hardware the X-pire 2.0-Viewer is running on, for example, to extract secrets from the trusted hardware.
- **not** capable to access the device of another user on which the X-pire 2.0-Publisher is running and is, therefore, not able to steal keys during the publication phase.
- **not** capable of compromising the X-pire 2.0-Keyserver and the PKI infrastructure used.

3.4.3 Background on Trusted Computing

The key idea behind trusted computing is to provide a software and hardware environment (usually referred to as trusted execution environment) with a solid root of trust for the execution of further computations. This root of trust is commonly provided by a third party, which could be everything from a hardware component up to an external (Web) service: It depends on the requirements and assumption for a particular scenario,

i.e., what is actually needed, what or whom are we going to trust in a particular scenario, etc. A common assumption for roots of trust in hardware (for example, a key stored in a dedicated chip) is, for example, that the hardware component cannot be compromised or that a key cannot be extracted by so-called lab attacks[28].

The usual goal behind trusted computing is either to achieve a trustworthy execution of a function or service (i.e., to have a trustworthy and correct output), or to execute a function or service on trustworthy input. Another recent goal of trusted computing is to achieve a robust protection of secrets by outsourcing the data to, or even directly generating it by the trusted third party.

All of the mentioned goals behind trusted computing share the major requirement that the hardware and software environment used needs to be set up trustworthy, which led to the development of concepts and techniques for a trusted boot process, memory and storage protection/separation, or the (remote) attestation of a trustworthy system state at runtime. *Trusted Boot* (also referred to as *Secure Boot*) approaches [219, 235, 13] aim at a fully trustworthy boot chain starting from the BIOS, going over the actual start-up code (including firmware loaders and firmwares as well as operating system loaders) and leading up to starting the actual operating system kernel or virtual machine managers (VMMs) [95]. The protection or separation of memory (RAM) and storage (flash) is usually achieved by enforcing the separation at the hardware level through dedicated chips or as part of modern *System-on-a-Chip* designs.

Popular hardware concepts that enable *Trusted Computing* include Trusted Platform Modules (TPM) that follow the specification through the Trusted Computing Group (TCG) [224, 224, 223], Intel's Trusted Execution Technology [95], and ARM's TrustZone technology [13].

3.4.3.1 Trusted Platform Module

The concept of Trusted Platform Modules (TPMs) was developed by the Trusted Computing Group, an industry consortium (members are for example the chip manufacturers AMD, ARM, Atmel, Infineon, or Intel) that describes its own task as "develop and support open industry specifications for trusted computing across multiple platform types" [224]. The TCG was founded in 2003 and coined the term of *Trusted Computing*. The standard for TPMs that is currently most used is version 1.2 [222] (version 1.2 became also an international standard ISO/IEC 11889 [225]), but its successor functionality has already been released in form of a TPM 2.0 library specification [223].

The TPM standard defines a minimum set of functionality that a TPM needs to fulfill, but it may integrate further functionality. In general, Trusted Platform Modules constitute dedicated hardware chips that integrate cryptographic algorithms and storage facilities. According to the TPM specification 1.2 [222], a TPM requires, for example, the following components: a cryptographic co-processor providing a random number generator, asymmetric encryption and signatures based on RSA [188], hashing based on SHA-1 [119], as well as a keyed-hash message authentication code (HMAC) [97], a key

[28] Lab attacks are advanced hardware attacks. Attackers try, for example, to intercept communication channels at circuit level on the physical boards, to slice and analyze single chips, or to use side-channels when hardware is executed outside its specifications (for example, for cold boot attacks when the temperature of components is lower than specified [98].

generator, a unit that detects changes in the power states of a system, an execution engine that executes the TPM commands received via the I/O ports, or platform configuration registers (PCRs) to store integrity measurements. The provided components allow, for example, the runtime attestation of systems or the generation and storage of keys for a disc encryption.

3.4.3.2 Intel Trusted Execution Technology

The Trusted Execution Technology (TXT) was invented by Intel and is supported by most of Intel's current CPUs. It provides a so-called *Measured-Launch-Environment* (MLE), which allows the comparison of the state of critical components with a known good state [95]. In particular, Intel TXT provides *Verified Launch* for booting MLE into a good state, *Launch Control Policy (LCP)* to document approved code, *Secret Protection* to prevent data leakage whenever MLE is not correctly shut down, and finally, *Attestation* to allow local or remote users to verify the good system state at runtime [95]. In order to function, Intel TXT requires two third-party components: A Trusted Platform Module 1.2 and an Intel TXT-enabled BIOS [95]. Altogether, these components set up a trusted execution environment (TEE).

3.4.3.3 ARM TrustZone

ARM provides with the TrustZone a security framework for many of its SoCs designs [4, 13] that provides all capabilities to implement a trusted execution environment as specified by GlobalPlatform [164]. The major concept behind the TrustZone is the introduction of a so-called *secure world* and a *normal world* mode. TrustZone partitions all hardware resources to provide each world with its own set of resources. The normal world can initiate a switch (possibly after authentication) into the secure world and vice versa. However, system resources such as RAM or persistent storage can be protected so that, for example, the normal world cannot read confidential data from the secure world. Inside the processor cores, the separation is achieved in a different manner: instead of partitioning the processor, processor cores supporting TrustZone can securely execute normal and secure code on one single physical core in a time-sliced manner [13]. Furthermore, it is also possible to show content from the secure world on the display without the normal world being able to access it. Implementation-wise, TrustZone could be used either by implementing a full-fledged secure OS for the secure world, or by executing only a library inside the secure world [13]. For the latter, it is usually mandatory to execute calls from the normal world OS, which requires robust integrity checks upfront.

Although TrustZone provides a powerful security framework for existing ARM-powered devices such as, for example, smartphones or tablet computer, it is currently out of reach to implement software relying on it, since there are currently no open and well documented devices available. Consumer devices have fully locked the TrustZone functionality for normal application developers. Nevertheless, ARM's TrustZone concept has been subject of several research projects. Winter presents a virtualization framework and discusses its integration with the TrustZone technology and the Linux kernel [237]. Based on this framework, Winter further presents a prototype of a mobile trusted

platform design. In [238], Winter et al. present a development platform for the TrustZone that is based on the QEMU emulator [27]. Wan Hussin et al. combine the Symbian DRM system and the TrustZone to implement an electronic entry pass [109]. Liu et al. leverage the ARM TrustZone for providing trustworthy sensor input to trusted mobile applications [134]. Recently, Santos et al. showed how to build a trusted language runtime based on mobile applications [192].

3.4.3.4 Trusted Computing and X-pire 2.0

In the context of X-pire 2.0, trusted computing is required to shift the processing and storage of protected images and keys to a trusted third party (the TrustZone components of the device used for viewing images), since the user and the *normal world* operating system are not considered trustworthy. By moving this processing into the *secure world*, we can prevent a leakage of decrypted data or keys. We would like to emphasize that we do not in general suspect the average user, however, an attacker can take the role of a user and interact with X-pire 2.0 with malicious intent. This includes the interaction with X-pire 2.0 while having root privileges in the normal world OS.

3.4.4 System Details

In this section, we describe the technical details of X-pire 2.0. We start with a detailed description of our communication protocol, both for the *Publication Phase* and the *Viewing Phase* (cf. Figure 3.20 and 3.22 for a detailed pictorial overview of both phases), and explain afterwards in detail our solution to the DDP, the integration of X-pire 2.0 into existing infrastructure, and the profiling protection at the X-pire 2.0-Keyserver to protect the viewer's privacy. Although X-pire 2.0 resembles a generic framework for providing a digital expiration date for any data, the integration into the existing infrastructure with possible post-processing needs to be adapted for the target publication platform. Therefore, we focus in the following (also for the protocol description) on the most prominent use-case for such an integration: uploading protected JPEG images including an enforceable expiration date to social networks such that they survive the post-processing.

3.4.4.1 X-pire 2.0 protocol

Publication Phase. The publication phase starts with the X-pire 2.0-Publisher creating an image img for publication (1). Afterwards, the X-pire 2.0-Publisher application (2) generates an image-specific symmetric encryption key k for the image img. This key is now used (3) to encrypt the image img gaining the encrypted image c_{img}. The encryption is followed by hashing the encrypted image c_{img} (4) to $hash$, which later serves as identifier for the encrypted image. Subsequently, the encrypted image c_{img} is embedded together with the X-pire 2.0 version information ver, the address of the keyserver ks, and the key identifier id_k (required to identify the key for the decryption of c_{img}) into a standard compliant JPEG container image $cont$ (5). After the ciphertext c_{img}, ver, ks, and id_k have been embedded into the container JPEG $cont$, we refer to this file as emb_c. The container image with the embedded ciphertext emb_c is now

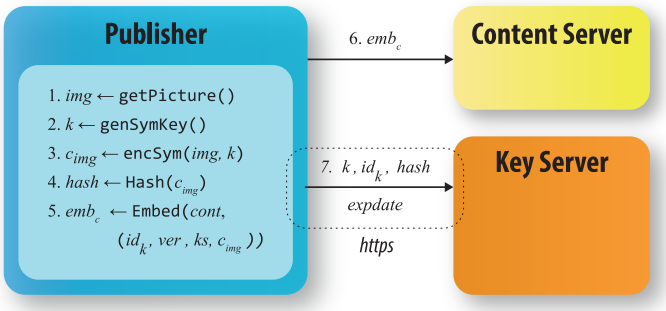


Figure 3.20: X-pire 2.0: technical details of the publication phase

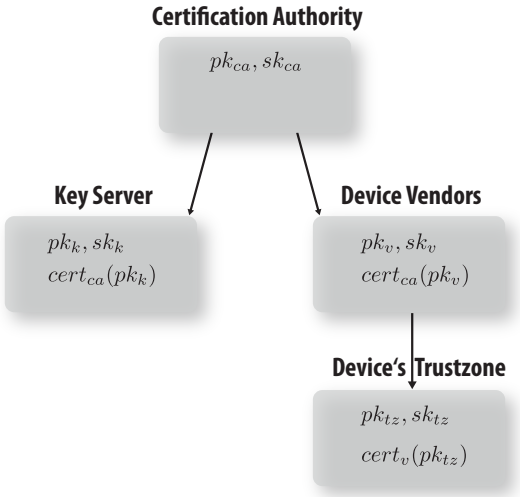


Figure 3.21: PKI hierarchy

uploaded to a content server (6) where it is published on the Internet. Once emb_c is published, the encryption key k, the key identifier id_k, and the hash $hash$ of the encrypted image c_{img} are uploaded in a final step to the X-pire 2.0-Keyserver using a secure HTTPS connection (7). After the publication phase has been finished, the user can log in to the X-pire 2.0-Keyserver during the *Update Phase* to manage previously uploaded keys, for example, to shorten or prolong existing expiration dates of keys, or to enforce the instantaneous expiration of a key.

Viewing Phase. During the viewing phase, the user opens a Web page w (1) that is checked for X-pire 2.0-protected images emb_c (2). In case such an encrypted image emb_c is detected (3), the X-pire 2.0-Viewer app for the *normal world* OS forwards the image to its counterpart inside the device's TrustZone (4). The X-pire 2.0-Viewer inside the TrustZone (X-pire 2.0 TZ Viewer) extracts now from emb_c the encrypted image c_{img}, the key identifier id_k of the key required for decryption of c_{img}, the address of the X-pire 2.0-Keyserver, and the X-pire 2.0 version information (5). After hashing c_{img} and gaining

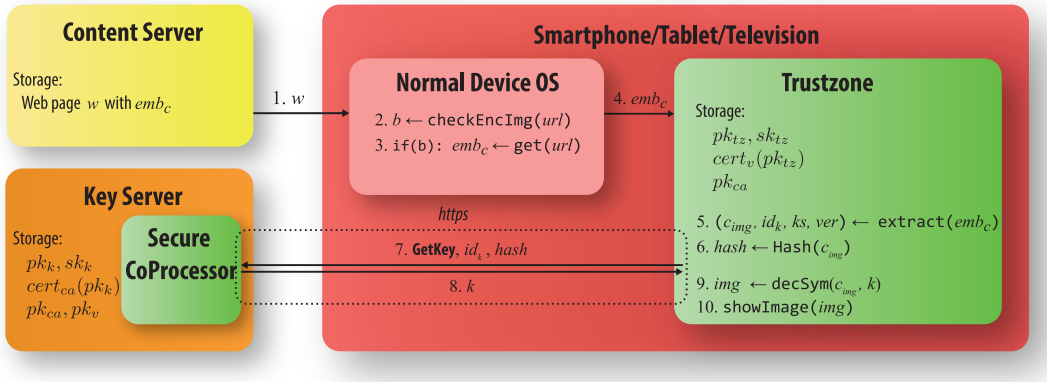


Figure 3.22: X-pire 2.0: technical details of the viewing phase

$hash$ (6), this hash and the key identifier id_k are used to request the corresponding decryption key k from the X-pire 2.0-Keyserver via a mutually authenticated HTTPS connection to the secure co-processor of the keyserver (7).

Both the X-pire 2.0-Keyserver and the X-pire 2.0 TZ Viewer need to authenticate themselves based on the existing PKI using their public key certificates ($cert_{ca}(pk_k)$ and $cert_v(pk_{tz})$, respectively). This two-way authentication is essential for two reasons: the X-pire 2.0 TZ Viewer needs to ensure that it is connected to the correct keyserver in order to prevent man-in-the-middle (MITM) attacks. Additionally, the X-pire 2.0-Keyserver needs to ensure that it is indeed communicating with an eligible TrustZone, since otherwise the decryption key k could be requested by an untrusted party to create digital copies of keys or images (after decrypting them). To achieve the mutual authentication, the X-pire 2.0-Keyserver stores all public vendor keys pk_v (for example, from Apple or Samsung) that provide X-pire 2.0 TZ Viewers running inside eligible TrustZones in their devices. For each public key of a vendor v, it exists a certificate $cert_{ca}(pk_v)$. The vendors use the secret keys sk_v corresponding to their public key pk_v to sign each device-specific TrustZone key $cert_v(pk_{tz})$, so that each device has a unique key, but still can be identified by the X-pire 2.0-Keyserver. We present the signing hierarchy of our PKI in Figure 3.21 and mark in the overview (Figure 3.22) of the *Viewing Phase* the keys and certificates stored by each party involved.

After the X-pire 2.0 TZ Viewer, which is running inside the TrustZone, has received the decryption key k (8), it decrypts c_{img} to img (9). Finally, the image is shown on the screen by the X-pire 2.0 TZ Viewer that is executed inside the TrustZone (10). By using the capability to directly display content on the device screen via protected access to the required hardware, it is ensured that the untrusted *normal world* OS has no access to the displayed content and, thereby, cannot make any digital copy (even not a screenshot) of the protected content.

3.4.4.2 ARM TrustZone for solving DDP

In order to solve the DDP, we need a technique that allows us to request the decryption key, to store the decryption key itself, and finally also to display the decrypted content in

a trusted environment inaccessible for the *normal world* OS, benign users, and attackers. This ensures that benign users, attackers, and even the *normal world* OS cannot access any of the involved sensitive information (decrypted data, keys) in a digital form, which renders also screenshots impossible.

The ARM TrustZone framework provides us with the required technical basis. X-pire 2.0 uses the TrustZone to protect access to decryption keys retrieved from the X-pire 2.0-Keyserver and to decrypted content. Additionally, the TrustZone maintains the secret key sk_{tz} that is used for the mutual authentication of the keyserver. All this sensitive information is only stored within the secure world. After the X-pire 2.0 TZ Viewer retrieves data for decryption, it requests the decryption key over a secured HTTPS connection from inside the TrustZone, such that we have a secure communication channel between the TrustZone and the keyserver. Furthermore, X-pire 2.0 executes the decryption process solely inside the secure world. Finally, the TrustZone framework is also used to directly display the content on the device's screen by following the approach described in [13], so that the normal world is not able to access displayed content.

In order to improve usability in terms of runtime, the X-pire 2.0 TZ Viewer could cache decryption keys, which prevents repeated key requests for resources that have been viewed before. This feature requires additionally a tamper-resistant hardware clock to ensure that keys are not longer cached than the expiration date allows.

3.4.4.3 JPEG Integration

X-pire 2.0 relies on the same JPEG integration mechanism as X-pire!. The mechanism constitutes an integral part of X-pire 2.0 and was integrated without further modifications. For details on the robust embedding procedure, please refer to the detailed description in Section 3.3.4.4.

3.4.4.4 PIR-based Profiling Protection

X-pire 2.0 involves two servers: an external content server and the keyserver. The content server learns less information than in the setting without X-pire 2.0, since the server itself cannot decrypt content. The keyserver, however, follows a centralized approach and introduces, therefore, the possibility of creating detailed viewing profiles of users. Since we require a central keyserver for our security guarantees, we need to overcome this drawback. Therefore, we propose the usage of private information retrieval (PIR) to protect the viewer's privacy by ensuring that no profiling occurs. The high-level idea of PIR is that, although all keys are stored in a database on the X-pire 2.0-Keyserver, the keyserver does not learn viewer requests and resulting key responses. In 2008, Williams and Sion [236] presented the first approach to PIR that, in terms of practicability, is computationally fast enough to be deployed in a real-world system. Their approach relies on an ORAM construction [87] that is based on secure hardware, such as the 4765 cryptographic co-processor by IBM [110]. The utility of such an approach has also recently been presented in a slightly modified setting to achieve privacy preserving online advertisements [20]. The PIR approach presented by Williams and Sion constitutes a promising approach for a future realization that hides access

Figure 3.23: Example for a typical publication device (left, Samsung Galaxy Camera) and a typical viewing device (right, Google Nexus 7).

to the key database from the keyserver itself. Additionally, it is necessary to establish a mutually authenticated secure channel between the client-side Trusted Execution Environment and the secure co-processor to protect against network attackers and to hide all viewer requests and resulting key responses.

3.4.5 Implementation

We have prototypically implemented the client-side applications of X-pire 2.0 for Android. The client-side implementation consists of two independent Android apps, the *X-pire 2.0-Publisher* for the *Publication Phase* and the *X-pire 2.0-Viewer* for the *Viewing Phase*. The X-pire 2.0-Keyserver is implemented as a Web application with a database back-end. The prototypical implementation for the TrustZone components is, due to missing open and well-documented hardware, only available as a Java library that provides the interface of the TrustZone functionality in software[29]. However, we argue that the described functionality can be easily achieved with support of a vendor, because the ARM TrustZone framework supports all functionality required.

3.4.5.1 JPEG Embedding: The X-pire!-Library

The X-pire 2.0-Publisher and the X-pire 2.0-Viewer share the X-pire! native library implemented in C and C++, which provides the functionality for creating protected images and extracting them again. Its functionality includes the symmetric encryption and decryption of images based on the OpenSSL library, hashing based on the OpenSSL library, our embedding routines in conjunction with libJPEG, as well as a custom implementation of Reed-Solomon codes for the error-correction. For details on the implementation of the X-pire!-Library, please refer to Section 3.3.5.1.

3.4.5.2 The X-pire 2.0-Publisher

The X-pire 2.0-Publisher was implemented by Stephan Lukas as an Android application for Android 4.1.1. It can be directly used to capture images with the integrated camera[30]. After taking images, they are encrypted and embedded to a container image and the

[29] Obviously, this implementation in software cannot provide the desired security guarantees.

[30] Besides using Android driven cameras such as the Samsung Galaxy Camera (cf. Figure 3.23), of course also smartphones and tablet computers with built-in cameras can be used.

key is uploaded to the X-pire 2.0-Keyserver. Once the account information for the X-pire 2.0-Keyserver is added, this process is implemented as a one-click solution just as for normal camera applications. After the captured image has been encrypted and embedded by X-pire 2.0, it can be uploaded to the Web service desired by the publisher.

3.4.5.3 The X-pire 2.0-Viewer

The X-pire 2.0-Viewer is also implemented as an Android application for Android 4. It provides a Web browser interface by extending the existing `WebView` class of the Android framework[31] and uses a Java mockup for the required TrustZone functionality that interfaces the X-pire! C and C++ native library. The `WebView` allows to intercept Web requests by overriding the method `shouldInterceptRequest()` and, thereby, to replace the connection handling by a custom implementation that takes care of the detection of X-pire 2.0 protected images. The X-pire 2.0-Viewer app provides the required functionality and interfaces, but is currently only executed in the *normal world* operating system. For a future implementation running on a tiny OS inside the TrustZone, we would basically require the C and C++ library for all encryption/decryption and embedding/extraction tasks as well as network functionality and protected storage to be moved to the TrustZone.

Furthermore, we need a secure path to the graphics controller and screen to display content securely. In order to provide the highest convenience for users, we consider the following three different possibilities to display images on the screen:

Overlay via TrustZone. Whenever the browser inside the *normal world* loads an encrypted image, the X-pire 2.0-Viewer forwards the encrypted image to the X-pire 2.0 TZ Viewer inside the *secure world*, which overlays the decrypted image on the display signal sent from the *normal world*. Thereby, the decrypted image visually replaces the encrypted image inside the browser. However, this approach is currently infeasible, as it requires either a fast switching between the *secure world* and the *normal world*, or the possibility to execute code from both *worlds* in parallel.

Browser in TrustZone. An alternative approach would be to run a complete browser inside the TrustZone that includes all image decryption and processing functionality. However, this approach has two major drawbacks: The trusted code base running inside TrustZone would be highly increased, while most of the browser's features are actually not required. Additionally, it would need to be ensured that nothing executed inside the browser (for example, JavaScript), is able to access or even leak sensitive information.

Switch to TrustZone. This solution is a hybrid approach: Whenever encrypted images are loaded by the Web browser inside the *normal world*, the X-pire 2.0-Viewer queues these images and forwards all encrypted images to the X-pire 2.0 TZ Viewer (the counterpart running inside TrustZone) upon request. The X-pire 2.0 TZ Viewer

[31] The `WebView` class of the Android framework is meant for loading and rendering Web content inside of applications.

subsequently decrypts all images and displays them in a slide show running inside the TrustZone. This is the approach we follow in our mockup implementation.

3.4.5.4 The X-pire 2.0-Keyserver

The keyserver is implemented as a Web application with an SQL database back-end. Publishers can access the Web interface on an account basis and manage their existing key/expiration date portfolio. X-pire 2.0-Viewers have to use the TrustZone to interface the secure co-processor inside the X-pire 2.0-Keyserver and the connection is based on a mutually authenticated HTTPS connection. The Web application allows publishers, in particular, to:

- create new X-pire 2.0-Publisher accounts,
- add new $k_{id_{img}}$ (keys for encrypting data) linked to an existing X-pire 2.0-Publisher account,
- set and change the expiration date for uploaded keys $k_{id_{img}}$,
- delete keys (instant expiration of data).

X-pire 2.0-Publishers, identified by username and password connect to the keyserver via HTTPS to prevent man-in-the-middle attacks. To enable the publisher to easily identify all $k_{id_{img}}$ linked to the account, X-pire 2.0 allows to set an individual title for each key.

3.4.6 Evaluation of X-pire 2.0

In order to evaluate the performance of X-pire 2.0, we measure (1) the time it takes to encrypt and embed images, (2) the time for loading a static Web site with and without X-pire 2.0 image-checks enabled, and, finally, (3) the time it takes to decrypt and view protected images. The first and the second measurement do not involve calls to X-pire 2.0 modules running inside the TrustZone, but the third measurement would normally be executed inside the TrustZone. Our measurements were mainly performed on a Google Nexus 7 (1.2 GHz Quad-Core, 1GB RAM, Android 4.2.2.). Since the X-pire 2.0-Publisher is designed for an image capturing device, we also evaluated the performance of the encryption and embedding routine on a Samsung Galaxy Camera (1.4 GHz Quad-Core, 1GB RAM, Android 4.1.1.).

3.4.6.1 The Embedding

In order to evaluate the performance of the encryption and embedding routine, we measured the time it takes to encrypt and embed a given JPEG image from the storage and to store the final JPEG image again: on the Nexus 7 the whole process takes about 0.95 seconds, on the Galaxy Camera about 0.7 seconds. These results include the time for creating and uploading the image decryption key. The full process of taking a picture with the camera, encrypting and embedding the image, uploading the key, and finally storing the image takes about 2 seconds on average.

Number of Images	1	10	20	30
Check deactivated (sec)	0.28	0.31	0.38	0.59
Check activated (sec)	0.32	0.94	1.59	2.05

Table 3.3: Total time for loading Web sites

3.4.6.2 Client-side Performance

Once images have been integrated into Web sites, the X-pire 2.0-Viewer needs to check viewed Web sites for X-pire 2.0 protected images and to queue them for viewing inside the TrustZone. Our evaluation compares the time it takes to perform this check for a different number of images on Web sites with the time it takes to load these Web sites without the checks being activated (cf. Table 3.3). We would like to stress that this check does not need to be always active. One could, for example, deploy whitelisting and explicitly activate the check for Web sites containing X-pire 2.0 protected images only. Furthermore, we measure the decryption time for queued images, i.e., the time it takes from requesting the images to be viewed until all decrypted versions are actually available. This task takes slightly less than 0.5 seconds per image and scales linearly. The last measurement is slightly artificial, since, as discussed earlier, TrustZone-enabled and programmable hardware is not openly available. Therefore, our measurements were performed on the Java implementation only (did not include any TrustZone usage). However, we argue that this should not introduce a big difference: method calls inside the TrustZone are executed in a time-sliced fashion on the same physical CPU core as normal world executions are. Thus, CPU-wise there is no difference. There is some additional time needed for the context-switch into the TrustZone, but this is countered by the fact that the code running inside the TrustZone is much more lightweight than a full OS and could be highly optimized for the target use-cases. Overall, we do not expect a slowdown for the decryption task inside the TrustZone.

3.4.6.3 Server-side Performance

We did not perform a comprehensive performance evaluation for the X-pire 2.0-Keyserver, since we did not implement the private information retrieval (PIR) and only proposed to use it in Section 3.4.2. Without the implementation of PIR based on oblivious RAM, we refer to the performance measurement of the X-pire!-Keyserver (cf. Section 3.3.6.2), which can be seen as a realistic benchmark for a market ready implementation.

3.4.6.4 Security Evaluation

In the following, we discuss potential security breaches that would break our system and argue why they are not possible.

Decryption key becomes available to public: Within X-pire 2.0, the decryption key is only known to three entities, which are considered trustworthy: the publisher, the keyserver, and the Trusted Execution Environments in the viewing devices. The

publisher has no incentive to make the key publicly available since this would break the protection of her own data. For the keyserver, we assume that nobody compromised the server, so access could have happened only from the Trusted Execution Environment or the network communication. The communication channels are mutually authenticated and protected via HTTPS, so a network attacker cannot retrieve the key. Accordingly, a key leakage could have only happened at the Trusted Execution Environment, but keys never leave the Trusted Execution Environment after the retrieval and its code base is considered secure. Furthermore, according to our assumptions, hardware attacks are also not possible. Therefore, decryption keys cannot become available to the public.

Protected content becomes available in clear: Only two entities of the system get access to the *cleartext* content. The publisher has, just as for the key, no incentive to leak the data. The Trusted Execution Environment holds the *cleartext* content only for displaying it and never reveals it. Therefore, protected digital content cannot become available in clear.

Keyserver creates detailed viewing profiles: We propose the usage of PIR so that the keyserver is not able to learn which user actually viewed which content. Therefore, creating detailed viewing profiles per user is not possible.

3.4.7 Discussion and Limitations

X-pire 2.0 provides a comprehensive concept for a digital expiration date and user-controlled publications with strong security guarantees. Neither an attacker, nor a curious user is able to gain any access to keys and decrypted content. The biggest problem at the moment is the missing availability of openly programmable and well-documented TrustZone-enabled devices. Even non-consumer devices for developers are still missing. With the support of device vendors, a system like X-pire 2.0 would be easy to deploy. Our X-pire! and X-pire 2.0 systems prove that it is possible to deploy the systems even within the existing infrastructure without requiring an active collaboration. Still we would like to mention that active measures against X-pire!/X-pire 2.0 can efficiently destroy our current integration techniques for images. So if a provider of a Web service wants to actively prevent X-pire!/X-pire 2.0 on the Web server, even after images have been published, this is clearly possible. But it would not destroy the security of the X-pire!/X-pire 2.0-protection, but merely constitute an instantaneous expiration of the image data. Regarding the concept of the X-pire!-Keyserver, we discussed several approaches for storing the keys and decided for a dedicated keyserver infrastructure, since we believe it provides the best trade-off between functionality, security, and user privacy. We considered a decentralized keyserver infrastructure as used by Vanish [82], a dedicated single central keyserver, and finally, dedicated keyservers with a key-sharing scheme. We prefer the latter two approaches in conjunction with PIR to provide a maximum flexibility regarding the expiration date, but in general our embedding techniques would work with all of the mentioned approaches. There is no "one size fits all solution", but we think one of the three schemes will provide a valid solution for most of the requirements by publishers.

3.5 Related Work

In the following, we discuss related work of X-pire! and X-pire 2.0. The first group of approaches adds an expiration date to data by following the paradigm of encrypting data with a symmetric key and restricting access to these keys afterwards. We stress that none of these approaches has considered the DDP, i.e., the duplication of content and keys before their expiration. Moreover, none of them targets scenarios where published data is subsequently manipulated as, for example, done by image upload routines of social networks. The initial approaches that followed the paradigm of encrypting content aimed at securely deleting data (including copies) in archives. The systems focus on corporate use and come with different requirements and design principles that are incompatible with our setting: all storage servers involved know that they are actually handling encrypted data, post-processing of data is not supported, an adversary grabbing all keys while the data is available is not considered, and in some proposals the keyserver additionally aids with decrypting the data.

The first such system we are aware of is [31], which provided the basic principles. This system by Boneh and Lipton aims at removing files both from the file system and from backup tapes. Their motivation was the fact that frequent data backups relying on tape-based backup systems store the backup data on many different media, which made a later manual deletion quite tedious. Even with tape-libraries that automate the media selection and mount procedures of single tapes, the deletion of single files would have take quite some time. Therefore, they pioneered with the idea of deleting information without actually touching the involved storage media, but by using cryptography. Their system encrypts a collection x of files with a specific key and configures an expiration data for that key as well as the maximum number of keys the system stores for this particular collection x of files. Whenever a key expires, it is added to the list of revoked keys and a new key is generated for the next backup of x. Once the list of revoked keys is full, adding a new one overwrites the oldest keys, which leads to an automatic expiration of backups that where encrypted with that key.

Another system following a similar concept is the *Ephemerizer* [169, 167] system by Perlman. It outsources the key management to a trusted third party called the Ephemerizer[32], which is responsible for creating, certifying, and publishing keys (the paper refers to them as ephemeral keys). Interesting is the idea behind Perlman's time-based scheme: all data for a particular expiration date is encrypted using the same key. The system supports a time-based scheme, a custom scheme with keys per file, as well as an on-demand scheme for the deletion of files. All three schemes can be used in the same file system at the same time. The Ephemerizer system was later improved by Nair et al. who detected and fixed a flaw in [169]. Nair et al. introduce an identity based encryption system and enable additionally fine-grained user settings per file [152]. In 2010 Tang et al. presented the *FADE* system [216], which transfers the core ideas of the Ephemerizer to the setting of cloud storage services such as Amazon S3. It provides the assured deletion of data in the cloud based on fine-grained data management policies.

[32]The Ephemerizer could potentially be distributed among several trusted third parties using a secret sharing scheme.

Vanish [82] follows similar lines as the previously described systems, but stores shares of keys in dynamic hash tables (DHTs), a data structure underlying P2P-networks. The DHT-based P2P network will by design stop replicating the key required for decryption after a certain time. Since the DHT-based P2P networks are subject to constant change, the replication stop of data items is hard to predict. Therefore, Vanish relies on a k out of n secret sharing scheme by Shamir [202], so that key shares required for decryption can be spread over the P2P network. This ensures on the one hand that an unpredicted replication
hand a single node with a single key share that exists longer than it is supposed to does not destroy the vanishing of data. Although relying on the replication stop resembles a nice passive expiration mechanism since the key vanishes automatically and, thereby, renders decryption impossible, this expiration mechanism constitutes also the biggest limitation. For common implementations such as the presented ones based on Vuze [231] and OpenDHT [161], replication stops too early to be useful (8 hours for Vuze and up to one week for OpenDHT). Although the authors suggest a solution for these very short time-outs, we do not consider an active, per user re-publication service as a solution in practice. Wolchok et al. [240] presented a so-called Sybil attack against Vanish and the authors of Vanish discuss defenses against such Sybil attacks in their follow-up work on *Cascade* [81].

We would like to stress again that the DDP is considered in none of these systems; similarly, images, post-processing, or compliance with the existing infrastructure for custom file formats in general are not considered or supported by these approaches. Another difference to X-pire! and X-pire 2.0 is that we decided to use a dedicated keyserver solution to provide flexibility for expiration dates. Moreover, using a dedicated and controlled keyserver allows us to implement the mitigation mechanisms for the data duplication problem for X-pire! as well as our solution to the data duplication problem in X-pire 2.0 (cf. Section 3.4.4.2). We would like to emphasize that our solution to the JPEG post-processing is applicable to Vanish's decentralized approach as well.

Two further approaches that share the core idea of Vanish to leverage an existing time-based vanishing of data in an existing storage infrastructure are *EphPub* [46] (formerly *EphCom*) by Castelluccia et al. and a recent approach by Reimann and Dürmuth [184]. Instead of using the DHTs of a P2P network, EphPub stores the keys required for decryption in the caches of DNS servers. EphPub follows also the idea of splitting keys among several storage locations. The approach by Reimann and Dürmuth derives key shares from different Web sites and bases the key expiration on the fact that Web sites change over time. Similar to Vanish, both approaches do not consider a post-processing of protected data and the threat of copying keys during their validity. Moreover, both systems have limitations regarding the specification of a particular expiration date. EphPub, for example, requires finding a large number of domains with the same time to live (TTL). The study by Castelluccia et al. shows that TTLs of more than 7 days are rather uncommon.

A real-world deployed system that shares the core motivation of X-pire!/X-pire 2.0 and its related scientific approaches is the instant-messaging app *Snapchat* [208]. It allows users to send pictures to friends where they can only be viewed for a few seconds before they are deleted. However, this system does also not provide any robust security

guarantees such as a solution to the data duplication problem. Recently, a new Web mail service called *ProtonMail* [176] came up in Switzerland. The service is still in the Beta phase, however, it lists the security feature of *Self Destructing Messages* with a reference to Snapchat.

The second group of solutions related to X-pire! and X-pire 2.0 aims at securing privacy-sensitive content published in social networks. The major difference to X-pire 2.0 is that these approaches rely on external trusted servers to store all the data, which we believe does not scale reasonably well given the vast amount of images published every day. One example is *FaceCloak* [136], which replaces content to be published in social networks such as Facebook or MySpace by fake data. The original data is encrypted and stored on a different server and the keys required for decryption are only provided to friends who are supposed to view the encrypted content. Very similar to FaceCloak is *flyByNight* [135]: both approaches aim at protecting the privacy of information published in social networks, but in contrast to the Firefox extension FaceCloak, flyByNight is implemented as Facebook application. *None Of Your Business* (NYOB) [96] is another Firefox browser extension with similar goals as FaceCloack, but it aims at encrypting and embedding textual information in an undetectable manner to Facebook. The system closest to X-pire!/X-pire 2.0 is the *P3* system by Ra et al. [178]. P3 aims at a privacy-preserving photo sharing and the authors try, similar to us, to embed a hidden photo in a JPEG image that is published to an online social network. The authors recognized the same header stripping during the post processing routines of the image upload as we did. However, in contrast to us they did not achieve a robust embedding into image data. They only embed the link to the location of the protected image, which is currently stored at a cloud storage provider.

Finally, our techniques for robustly embedding data within JPEG files resemble steganographic techniques [41]. However, existing steganography approaches (for example, [226, 177]) do not consider robustness against JPEG re-compression as applied in upload routines of social networks: their focus is on embedding information in an undetectable manner, which yields an insufficient data rate for our embedding. In contrast to steganography, watermarking approaches do consider JPEG re-compression, but only partially: they only need to recover sufficiently many bits to statistically identify the document, which neither suffices to fully recover encryptions, nor to embed large images. Therefore, we had to develop a new JPEG embedding routine that is able to cope with JPEG re-compression and supports much higher data rates.

4 AppGuard

Today, touch-enabled smartphones and tablet computers have become our daily companions. They provide comfort and utility both in our private and our business lives as they combine new feature-rich applications with the information and service that was provided before by computers, PDAs, feature phones, navigation devices, etc. in one single and compact device. But as shiny and comfortable the usage of these devices is, so big are also the security and privacy challenges introduced by these devices. We have had never before a device that included so much functionality and is at the same time always in close proximity to us. In this section, we introduce with AppGuard a powerful tool to enforce security and privacy policies on Android devices. AppGuard allows users to efficiently protect their privacy according to their needs on standard consumer devices without requiring modifications to the operating system, root privileges, or the like. Moreover, AppGuard allows users to mitigate vulnerabilities in third-party applications and the operating system. This section is based on the following publications: [P6, P5, P2, P10, P3, P4].

4.1 Motivation

The major breakthrough of touch-enabled devices came with the release of the first iPhone in 2007 [12], when Apple presented a touch enabled mobile phone with a highly user-friendly and intuitive user interface. Another major catalyst for the success of today's smartphones and tablet computers was in 2008 the introduction of the Android operation system by the Open Handset Alliance, an industry consortium led by Google [146, 10]. Key to the success of smartphones and tablet computers (referred to as mobile devices from now on) is the fact that they provide us with rich functionality and a highly comfortable usage both in our private and in our business lives. We simply had never before a device that includes so much functionality in a single device, especially not at this small size. These mobile devices fit into our pockets just as the feature phones

before, but the list of functionality they integrate is much richer. A classic feature phone supported phone calls, text messages, an address book, often a calendar and an alarm clock, and in the last years before the iPhone was introduced sometimes also audio players and a camera. Modern smartphones and tablet computers provide a full mobile office and more: They are always connected to the Internet and include e-mail support, a Web browser, synchronized address books, calendars and notes, multimedia players for audio and video, office applications for text documents, spreadsheets, and presentations, audio, image and video recording, applications enabled by new hardware sensors (e.g. GPS, accelerometers, etc.) such as navigation applications, fitness and health tracking applications, and many more. The list can easily be further extended by installing feature-rich applications from application stores (so-called app stores) on mobile devices to enable online banking, voice-over-IP communication, real-time usage of social networks (based on push notifications from these networks), etc. The list is endless and these devices can basically support everything known from desktop computers, feature phones, personal digital assistants (PDAs), photo and video cameras, multimedia players such as the iPod, navigation devices, and so on. With the recent introduction of health and fitness tracking applications based on special sensors, we can easily see that the development of new functionality for these devices will not come to an end soon.

But as shiny and comfortable the usage of these devices is, so big are also the security and privacy challenges introduced by them. We have had never before a device that included so much functionality and is at the same time always in close proximity to us. A classic feature phone has in comparison more or less no features. Laptop computers might include more features, but we do not always carry them with us. Mobile devices and, in particular, smartphones, are carried most of the time either in our pocket or our hands, or they are at least lying right next to us. We carry them with us in meetings, we have them during breakfast, lunch, and dinner, and during the night they are commonly lying on the nightstand. New is also the change of the data plans users usually have: Modern mobile devices are always online and allow their users to transfer more and more data within the default data plans. Both the proximity in which these devices are to their customers, and the permanent connection to the Internet make these devices the prime targets for many kinds of attacks. If an attacker manages to get remote access to a single device and to achieve extended or even root privileges, this basically allows the attacker to fully track a victim: the camera allows to transmit videos from the surrounding, the microphone allows to transmit all spoken words, GPS information allows to locate the victim, and accelerometers of devices in the pocket allow to infer current actions such as running or walking up stairs. Once the device is hacked, the attacker could simply access all data on and available to the mobile device. So the impact of a potential hack is for mobile devices much higher than for any device before. Most commonly this sort of high-impact hacks are the result of a targeted attack and such attacks are rare since they do not scale very well and are quite cost intensive.

However, there are also ways to achieve a similar impact (depending on the type of operating system vulnerabilities) with a much better scalability: The most common attacks against today's smartphones are attacks based on malicious applications. Attackers could strive for disguising malware with an innocent looking cover story in order

to spy out users. Such attacks scale much more efficiently since attackers can simply submit their malware to the app stores for third party applications and hope to get around the application vetting processes of the stores, if such processes exist at all. Past incidents have proven that such applications spread quite fast [189]. In addition to these attacks with malicious intent, a vast amount of applications for mobile devices is overly curious and breaches the users privacy: These applications try to access information that is not essential for their main functionality. The Twitter app, for example, accessed the users' contact list on smartphones as part of the "find friends" feature and stored the contact data on their servers for about 18 months [193]. With a similar intention also the social network apps Path and Hipster uploaded the user's address book to find friends already using the networks more efficiently [239]. Also the popular WhatsApp sends out all phone numbers from the user's address book and Android users, in contrast to iOS users, cannot opt-out [234].

Prime target for attacks on mobile systems is currently the Android operating system, which is easy to explain with its market share for smartphones and tablet computers. It achieved, for example, in 2013 a market share in sales of about 78,4% for smartphones [78]. But Android is also a thankful target for attackers: system updates to fix vulnerabilities are delivered for almost all phones (except for the Google Nexus series) at a very low rate, if at all. Moreover, updates are usually only provided for 18 month, which is problematic if the common contract duration with the mobile service provider is 24 month (e.g., in Germany). In order to prevent targeted attacks on Android (if possible at all), it clearly requires to harden the operating system and to include a sophisticated and comprehensive security concept for the system. The Android development started recently to move in the right direction by integrating security enhancements for Android based on SE Linux [207, 199] and integrating parts from Samsung Knox [179]. In case of malicious or overly curious software, the situation is slightly different: Although it is still challenging to detect malicious software based on static and dynamic analysis, we see a root cause for the success of such applications on Android in the bad and very inflexible design of the Android permission system. We introduce AppGuard to overcome these deficiencies of the Android permission system: AppGuard allows users to mitigate the impact of malicious[1] and overly curious applications on the users' security and privacy by protecting access to sensitive information.

4.2 Problem Description

The goal of achieving a powerful security and privacy protection for users against third-party applications on Android is challenging:

The operating system needs to integrate a mechanism that allows users to configure the access to sensitive information according to their needs. This is not the case for Android and a particular problem of the Android permission system, which does not allow any configuration of the access rights granted to third-party applications at all, neither at install time nor at runtime. Either users accept the access right (so-called permissions) requested by a third-party application during the installation process,

[1] AppGuard cannot prevent zero day exploits and several exploits based on native code.

or they cannot install the application at all. Integrating a new mechanism to the operating system implies that users would need to modify the operating system or at least temporarily gain root privileges, which is out of reach for laymen users. Moreover, it would most likely cause problems with, or even prevent future operating system updates. Since Android applications are by default unprivileged[2] and communication between two applications is only possible via Android's inter-process communication[3], it is also out of reach to install a third-party application that solves the job by controlling other applications. So it is necessary to develop a new policy enforcement framework that, one the one hand, overcomes the existing deficiencies of the Android system, but that one the other hand can also be installed on existing Android devices without negatively impacting the devices' functionality and without modifying the operating system or requiring root privileges. The framework requires a flexible policy specification mechanism that allows both the extension and the configuration of existing policies at runtime. Moreover, the policy specification should facilitate the specification of policies by security experts and enable in future the automatic generation of policies from even more high-level abstractions by users and from results of static and dynamic application analysis.

4.3 Contribution

AppGuard overcomes the deficiencies of the Android permission system and provides a powerful framework for the enforcement of security and privacy policies on unmodified Android phones. The system is based on inline reference monitoring and includes, in particular, the following contributions:

- AppGuard provides a flexible runtime enforcement for security and privacy policies on unmodified stock Android phones based on inline reference monitoring. The approach includes the full on-the-phone instrumentation of third-party applications (contribution by Philipp von Styp-Rekowsky, not part of this thesis).
- Security and privacy policies for AppGuard are specified in a high-level policy language called EXSPoX. The language describes security automata with the transformation capabilities of edit automata [132].
- AppGuard includes a policy-based approach to separate secrets. EXSPoX allows tagging data as confidential, which is subsequently enforced through AppGuard at runtime.
- Our system allows the specification of information flow control policies. In certain limits, even implicit flows can be prevented [P4].
- We prove the utility of AppGuard by applying our rich set of predefined policies in real-world use-cases to revoke Android permissions dynamically, to enforce

[2]Privileges can be extended by permissions, but Android does not include a permission to control other applications or to gain root privileges.

[3]Requires mutual cooperation by providing the required APIs.

user-desired security and privacy settings on third-party applications based on fine-grained and stateful policies, to enforce corporate policies, to mitigate in-app vulnerabilities, and finally, to mitigate vulnerabilities in the operating system.

4.4 Android Primer

Before we provide a detailed overview of AppGuard, we will first introduce the necessary background information on the Android operations system. Android was first released in September 2008 [146] and had in 2013 already a market share in sales of 78,4% for smartphones [78]. The system was introduced by the *Open Handset Alliance*[4], an industry consortium led by Google with the goal to promote Android as an open source operating system for mobile and embedded devices [10]. The current deployment version of Android is 4.4 (code name: *KitKat*) and the system is used in more and more use-cases. Android version 4.4 targets already at wearable devices, smartphones and tablet computers, cars, as well as TVs [6]. In order to achieve the quality of service desired by users for all four mentioned use-cases, Android systems typically integrate various hardware sensors to acquire precise high-quality input data for applications. However, this input is user-individual and contains lots of sensitive data (for example, location data, data from accelerometers, audio and video input, and in future probably also medical data from wearable devices or telemetric data from cars). So Android also needs a robust and comprehensive security concept that enables feature-rich applications on the one hand, and protects user-data on the other hand.

The Android systems itself consists of a multi-layered architecture (cf. Figure 4.1) that builds up on a standard Linux kernel. The Linux kernel integrates the actual device drivers, wakelocks, an aggressive memory management, and, finally, Android's inter-process communication mechanism: the so-called `Binder` [9]. On top, Android integrates several native system libraries (for example, `libc`, `SSL`, or `OpenGL`) as well as the Dalvik virtual machine with core Android libraries. The Dalvik virtual machine is register-based and replaces the standard stack-based Java virtual machine to improve performance on resource-constrained mobile devices at the cost of slightly larger bytecode [204]. The Android application framework provides developers with the Android API and thereby with access to a large variety of predefined classes and services (for example, location manager service or the activity manager service). All applications (both applications by vendors and third-party applications) run on top of the application framework.

Android applications are programmed in Java against the Android API provided by the application framework and may integrate native code and libraries via the so-called Java native interface. The Android API and Android's support for both Java and C/C++ code enables developers to implement fast feature-rich applications that integrate a variety of sensors available in the individual devices. Android applications themselves are made up of so-called components and the system integrates four different types: *Activities, Content Providers, Broadcast Receivers, and Services* [8]. In the following, we introduce the four different component types in detail:

[4]Details: `http://www.openhandsetalliance.com`

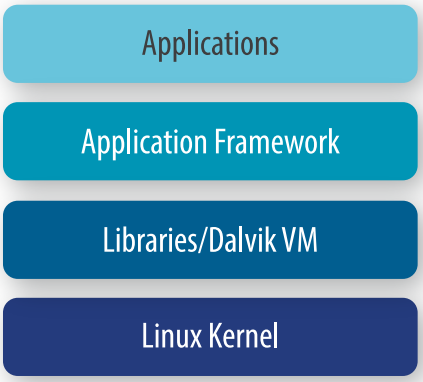


Figure 4.1: The Android Architecture: A High-level Overview (cf. (7))

- *Activities*[5] make up the typical application components for foreground tasks. They include a single screen for the user interface and *Activities* are the main parts of applications users can actually see and interact with.
- *Content Providers*[6] provide developers a generic interface for accessing structured data. The management of this data is achieved in an SQL-like manner.
- *Broadcast Receivers*[7] receive so-called *intents*, the message format for inter process communication in Android. *Broadcast Receivers* can be seen as typical mailboxes of applications: they receive broadcast messages from other applications (mostly status messages from system services) such as "Incoming call" or "Battery low" and allow applications, thereby, to react on a particular system state.
- *Services*[8] fulfill typical background tasks that do not require any user interaction (for example, a file download from the Internet in the background).

The security concept of Android is based on application sandboxing and the security mechanisms of the underlying Linux kernel. Every application is executed in a different process under a different Linux user ID (UID)[9] and, by default, without any further access rights. Applications can only access their individual data directories and unprotected system APIs. The access separation between applications is achieved based on the discretionary access control of the Linux kernel. In order to access protected resources, access rights of applications can be extended based on Android's permission system. When users install third-party applications, they can review the permissions requested by the developer. Afterwards, they can either accept and grant these permissions

[5]Details: http://developer.android.com/guide/components/activities.html
[6]Details: http://developer.android.com/guide/topics/providers/content-providers.html
[7]Details: http://developer.android.com/reference/android/content/BroadcastReceiver.html
[8]Details: http://developer.android.com/guide/components/services.html
[9]Developers can enforce that their different applications run under the same UID.

by installing the application, or users cannot install the application. Permissions are either enforced inside of Android services such as the location manager service, which is underneath reflected by the discretionary access control at the kernel level: access to devices is granted if a UID is member of the corresponding Linux group that holds the access, i.e., based on the group ID (GID). Or, permissions are directly enforced at the kernel level when applications are able to access hardware resources directly as done for network connections or standard file system access. This is directly based on the fact whether the calling UID is member of the required group (GID). Communication between applications is only possible via Binder and also subject to the permission system. Applications can expose custom APIs protected with custom permissions that need to be held by calling applications. In order to easily identify authors of third-party applications, all applications need to be further signed with a developer key, which also ensures secure application updates.

4.5 System Overview

AppGuard enables the enforcement of fine-grained and stateful security and privacy policies on unmodified stock Android phones. AppGuard follows the approach of inline reference monitoring pioneered by Erlingsson and Schneider with the *SASI* system in 1999 [68]. The concept of inline reference monitoring is central to the enforcement of policies on unmodified Android phones, since the security monitor needs to be directly integrated into third-party applications. We provide a high-level overview of our system in Figure 4.2. Starting point of every inlining process are untrusted third-party applications as well as a set of pre-defined security and privacy policies and the corresponding user configurations. The untrusted third-party application and both the policies and the configuration chosen by the user are input to the rewriting component of AppGuard. Afterwards, the rewriter inlines the required enforcement points for the corresponding policies into the untrusted third-party application. Once the security monitor is inlined and the application re-packaged, it can be installed just like any other third-party application. Once the secured application is installed, the security monitor enforces the security and privacy policies at runtime and the application is able to receive dynamic updates from the management application of AppGuard.

We see AppGuard as the major building block to protect the user's private data against overly curious and malicious third-party applications. Figure 4.3 provides the big picture of how our tools and techniques related to AppGuard can be integrated in the future to improve both the specification of policies as well as the actual security and privacy level for users through AppGuard. We aim at achieving an automatic generation of security and privacy policies based on a comprehensive analysis of third-party applications through *Bati* and plan to design a tool for the high-level specification of policies by users based on recent advances in the area of usable security and privacy.

4.6 System Details

This section presents the details of our AppGuard system. First, we present the details of the policy specification for AppGuard in our extended version of SPoX policy language

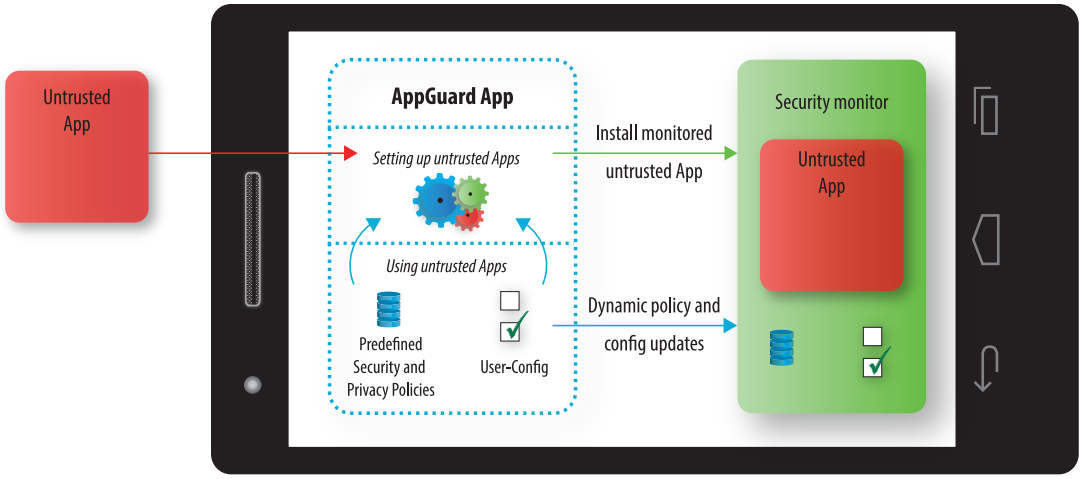


Figure 4.2: Schematic overview of AppGuard

in Section 4.6.1, before we continue with example policies in Section 4.6.2 and introduce the actual enforcement mechanism of AppGuard in Section 4.6.3.

4.6.1 Policy Specification

AppGuard leverages a high-level policy language to describe the intended security and privacy policies that should be enforced. Our policy language is based on the SPoX policy language as it was introduced by Hamlen et al. in 2008 [100, 101]. In order to fulfill our design goals, AppGuard requires a policy language with the transformation capabilities of edit automata as introduced by Ligatti et al. in 2005 [132]. However, the original SPoX language was designed to follow the concepts of aspect oriented programming and does not provide the transformation capabilities of edit automata. SPoX encodes a security automaton where nodes correspond to the current security state, and edges are labeled with so-called pointcuts, the matching patterns for security relevant functions in the code. For AppGuard, we also wanted to integrate a policy language that directly encodes a security automaton, but one with the transformation of an edit automaton. So, we decided to extend the SPoX language to fulfill these requirements of AppGuard. In order to achieve the transformation capabilities of an edit automaton, we added a statement for rewriting existing function calls in the original code (suppress and replace) to the original SPoX language, which gives us the transformation capabilities of an edit automaton. Furthermore, we added the possibility to tag return values of function calls as secret to enable the specification of our secret separation. We refer to our extended version of SPoX as EXSPoX.

Using EXSPoX, policy authors can now specify all method calls that should be monitored and define alternate control flows by rewriting methods. All method calls that are not present in the security automaton are also not subject to constraints enforced through our security monitor. Using our secrecy tagging, we can also define through the edit automaton defined by our EXSPoX policy how secret data can be processed

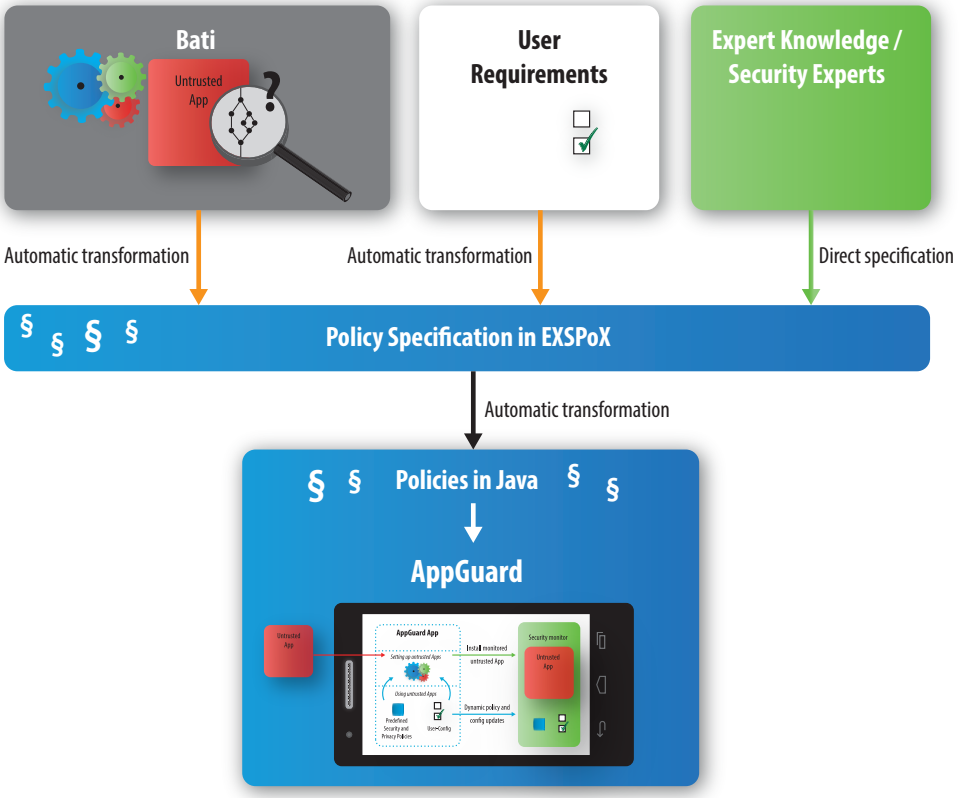


Figure 4.3: Policy Generation for AppGuard (P4) based on *Bati* (T1) and work by Gregor Geßner (T2), as well as Future Work (orange arrows)

and how not. The secrecy tagging allows us to reason about the confidentiality of data on the mobile device, and about data that is received as input, for example, via the microphone or via network connections. Although AppGuard does not integrate a dynamic taint tracking approach, our approach still allows us to specify information flow control policies. As long as we do not permit the usage of equality tests inside of application code, our system is even able to successfully prevent critical implicit information flows [P5].

We present the syntax of EXSPoX in Table 4.1 and 4.2, where we also explicitly mark our extensions. For the syntax of the original SPoX language, we refer the interested reader to [101]. The major extensions for the EXSPoX language constitute the support for rewriting function calls as well as the support for tagging return values as secret.

Tagging return values as secret is achieved by adding a few definitions to the policy syntax of SPoX (cf. Table 4.1). The `nodes` *oid* [+,-]-statement in Table 4.1 allows us to label the return value with the object identifier *oid* as secret. Alternatively, if *oid* is already labeled as secret, the statement allows us to remove such a secrecy label again. Adding or removing a secrecy label is handled by either adding or removing the *oid*

Table 4.1: EXSPoX policy syntax. Additions to SPoX are marked by †.

	$n \in \mathbb{Z}$	**integers**
	$c \in C$	**class names**
	$sv \in SV$	**state variables**
†	$oid \in O_{ID},\ O_{ID} \subseteq SV$	**object identifier**
	$x ::= c \mid oid$	**callee identifier**
†	$cid \in C_{ID}$	**call identifier**
	$iv \in IV$	**iteration vars**
	$en \in EN$	**edge names**
	$pn \in PCN$	**pointcut names**
	$pol ::= np^* sd^* e^*$	**policies**
	$np ::=$ (`pointcut name =` "pn" pcd)	**named pointcuts**
	$sd ::=$ (`state name =` "sv")	**state declarations**
	$e ::=$	**edges**
	(`edge name =` "en" `[after]` $pcd\ ep^*$)	edgesets
	| (`forall` "iv" `from` a_1 `to` $a_2\ e^*$)	iteration
	$ep ::=$	**edge endpoints**
	| (`nodes` "sv" a_1, a_2)	state transitions
	| (`nodes` "sv" $a_1, \#$)	policy violations
†	| (`nodes` oid `[+,-]`)	setting secrecy-level of object identifiers
	$a ::= a_1 + a_2 \mid a_1 - a_2 \mid b$	**arithmethic**
	$b ::= n \mid iv \mid b_1 * b_2 \mid b_1 / b_2 \mid (a)$	

label from the list `SI` that is responsible for tracking all objects currently labeled as secret. `SI` is modeled as a subset of the state variables. Whenever a state variable is changed, this introduces a new state which is reflected as a unique node in EXSPoX.

In order to add support for rewriting function calls in EXSPoX, we modified and extended the pointcut syntax of SPoX. The new pointcut syntax of EXSPoX is shown in Table 4.2, where we marked again the new additions. First of all, we modified the `call`-statement to include two new identifiers: an object identifier *oid* referring to the returned object of a method call and a call identifier *cid* for one concrete method call. Important to mention is that *oid* is an equivalence class for objects that are returned by a particular function call. This reflects that usually the return value of a particular function is treated the same way. If it needs to be treated differently, this is caused by special circumstances (for example, because the input is confidential) and in this case, this particular function call can be referenced by *cid*. Second, we added some completely new definitions, mainly to allow alternate control flows in case of policy violations. This includes the graceful reaction in case of policy violations, for example, to prevent applications from crashing if access to data is denied by returning mock values. The most important addition to achieve this is the `rewrite`-statement. The `rewrite`-statement specifies a series of function calls that replace an existing function call in case of a policy violation. Predicates that are specified inside the second argument of the `rewrite`-statement, i.e., for the new alternate function calls, are conditions that

Table 4.2: EXSPoX pointcut syntax. Additions and modifications to SPoX are marked by † and ‡, respectively.

	$re \in RE$	**regular expressions**
	$md \in MD$	**method names**
	$fd \in FD$	**field names**
	$pcd ::=$	**pointcuts**
‡	$\mid (cid : oid = \texttt{call}\ mo^*\ rt\ x.md)$	method calls
‡	$\mid (\texttt{argval}\ cid\ n\ vp)$	stack args (values)
‡	$\mid (\texttt{argtyp}\ cid\ n\ c)$	stack args (types)
	$\mid (\texttt{and}\ pcd^*)$	conjunction
	$\mid (\texttt{or}\ pcd^*)$	disjunction
	$\mid (\texttt{not}\ pcd)$	negation
†	$\mid (\texttt{rewrite}\ pcd\ pcd)$	rewriting
†	$\mid (\texttt{order}\ cid^*)$	call order
	$mo ::= \texttt{public} \mid \texttt{private} \mid \ldots$	**modifiers**
	$rt ::= c \mid \texttt{void} \mid \ldots$	**return types**
	$vp ::= (\texttt{true})$	**value predicates**
†	$\mid (\texttt{secret})$	secrecy predicate
†	$\mid (\texttt{isanonymous})$	privacy predicate
	$\mid (\texttt{isnull})$	object predicates
	$\mid (\texttt{inteq}\ n) \mid ((\texttt{intne}\ n)$	integer predicates
	$\mid (\texttt{intle}\ n) \mid ((\texttt{intge}\ n)$	
	$\mid (\texttt{intlt}\ n) \mid ((\texttt{intgt}\ n)$	
	$\mid (\texttt{streq}\ re)$	string equality
†	$\mid (\texttt{argeq}\ cid\ n)$	argument equality

need to be fulfilled by a concrete implementation of the particular policy. In addition to the introduction of the `rewrite`-statement, we modified both the `argval`-pointcut and the `argtype`-pointcut to include the call identifier. This is necessary to pose extra constraints on type and value of arguments for concrete calls inside of rewrite statements. Furthermore, we introduced the `order`-pointcut to allow the specification of a particular execution order of several function calls inside of the rewrite statement. Finally, we added the `secret` predicate to test for secrecy, the `isanonymous` statement to pose particular constraints on alternative return values (for example, mock values for the address book), and `argeq` *cid n* to test for the equality with argument *n* of the function call *cid*.

The intended workflow for the specification of EXSPoX-policies for AppGuard is shown in Figure 4.3. Central to our workflow is the specification of our high-level policies in EXSPoX. Developers or security experts can specify the security or privacy policies directly in EXSPoX. Additionally, we consider it of high importance to develop in the future a tool chain based on our static analysis framework for Android called *Bati* [T1], that allows to automatically generate EXSPoX policies for particular applications based on the findings during the analysis of applications. Furthermore, we consider it as future work to design usable interfaces that allow also laymen users to also generate completely new policies based on abstract descriptions.

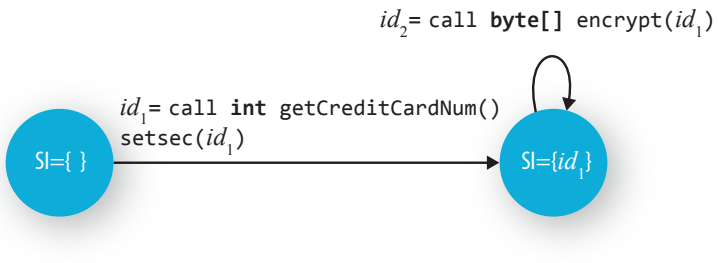


Figure 4.4: Automaton for a policy that protects the return value of a function that reads the credit card number.

Once the EXSPoX policies are automatically generated or manually written, they need to be transformed into the concrete Java implementation. The Java policies are implemented as standard Java classes and this format is also used during the inlining process of AppGuard. Similar to EXSPoX, the concrete Java implementations of policies are based on a direct mapping to security-critical API functions and include the corresponding decision logic for monitoring method invocations. But the concrete implementation of a EXSPoX policy in Java may vary, since the high-level policy language only poses requirements and constraints. A nice example is here the **isanonymous** statement for return values: it is up to the concrete Java implementation, whether the returned value constrained to a required type is completely random, or just not considered as personal information. If an application wants to access, for example, the address book and the application expects a value in return to function, most users would be fine with an empty entry, a random entry, or even an artificial but not fully random entry. An automatic transformation of EXSPoX-policies into a concrete Java implementation compatible with AppGuard is presented in the bachelor thesis of Gregor Geßner [T2].

4.6.2 Policy Examples

In the following we show the utility of EXSPoX on several examples. First, we will introduce the security automata that describe the policies and afterwards we introduce the corresponding EXSPoX policies. In case of our policy for rewriting critical HTTP connections by HTTPS connections we also present a concrete Java implementation of that particular policy for AppGuard.

EXSPoX allows us to specify policies that protect data against unauthorized access by labeling data as confidential. Moreover, EXSPoX allows us to specify a subsequent declassification of data through a particular function call (for example, by hashing or encrypting data). The declassification has no impact on the confidentiality of the input arguments of this function: they remain secret. An example of such a declassification is shown by the security automaton in Figure 4.4. The return value id_1 of the function call is set secret and id_1 is added to `SI`, which introduces a new configuration of the state variables, which is reflected by a new node.

The corresponding EXSPoX policy for the credit card example (cf. Figure 4.4) is

```
(edge name="ReadCreditCardNum"
    (and
        (cid1:id1=call "int getCreditCardNum()")
        (nodes id1 +)
    )
)
(edge name="EncryptCreditCardNUm"
    (cid2:id2=call "byte[] encrypt(id1)")
)
```

Figure 4.5: EXSPoX policy to enforce the secrecy of credit card numbers

provided in Figure 4.5. The first edge description sets, as in the automaton, the return value as secret. The second edge does not introduce a new state variable, since the return value of the encryption is, just like in the automaton, not tagged as secret. This can be seen as declassification since the return value is no longer considered as secret.

A similar, but slightly more complex scenario that frequently occurs in third-party applications for smartphones and tablet computers is the following: Applications such as WhatsApp or Twitter want to access phone numbers of the user's address book as part of their find friends features to automatically find contacts that already use the messaging or social network app. If users do not have the choice to opt-out and to add their contacts manually (as in an old version of the Twitter app [193]), its desirable to prevent this access. Figure 4.6 shows the security automaton that models the access to the address book and the required secrecy labeling based on the Android content provider for the address book.

The first function call in the automaton is to `CR.query(contacts, phoneNumber)`[10] and the returned `Cursor` object to the address book entry is tagged as secret. In the second step, either the cursor is moved forward, which has no impact on the state variables, or the phone number at the current position of the cursor is requested. The returned phone number is again tagged as secret. In the third state, again either the cursor can be moved forward, the phone number at the current position can be requested, or one can execute `getBytes()` on the returned `String` of the phone number. In the last state, the returned `byte` array of the phone number is hashed, which is considered as declassification since the return value is not tagged as secret. We added in state two the `moveToNext()` call as well as in state three both the `moveToNext()` call and the `getPhoneNumber()` call to provide a better overview and to illustrate the equivalence classes for the object identifiers `oid`. Since the return value of `moveToNext()` does not require any special treatment and the return value of `getPhoneNumber()` is already tagged secret, we could also omit these edges as they do not have any new effect on the policy enforcement.

The corresponding EXSPoX policy for the protected access to phone numbers of the user's address book is shown in Figure 4.7. Similar to the security automaton of this policy (cf. Figure 4.6), the edges `MoveToNode`, `MoveToNode2`, and `SecretPhoneNumber2` are only added for a better overview and could easily be omitted.

[10] `CR` is here the abbreviation for `ContentResolver`.

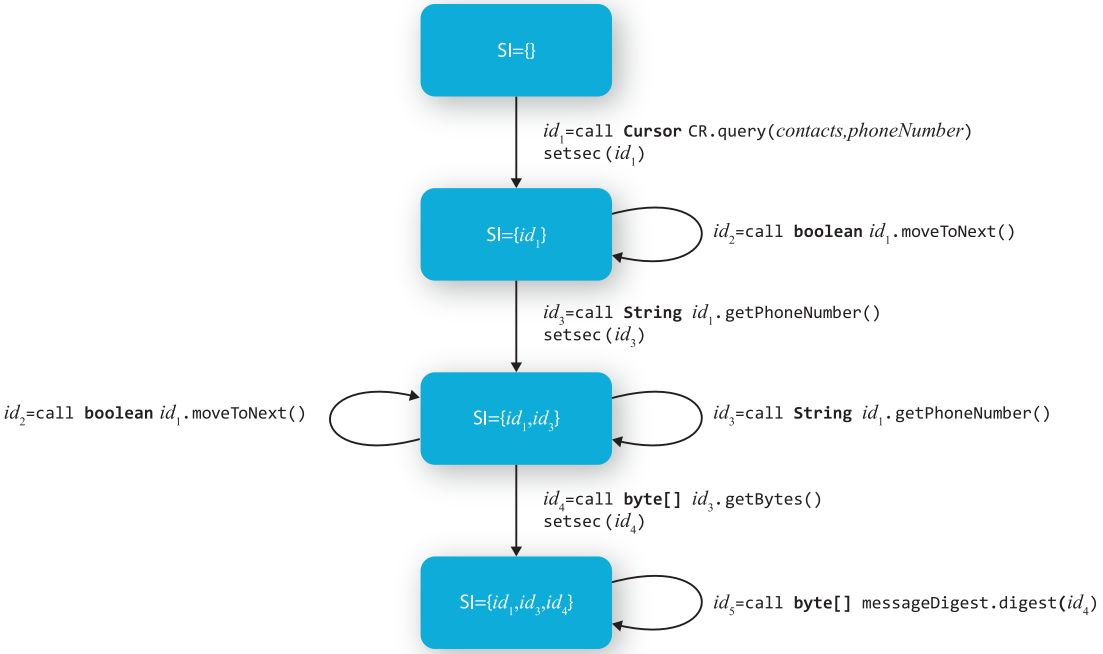


Figure 4.6: Automaton for a policy that describes the declassification of a phone number by hashing it.

Our last policy example aims at enforcing HTTPS for HTTP connections that are used to transmit confidential information (for example, session tokens). The security automaton that enforces the rewriting of such HTTP connections to HTTPS is shown in Figure 4.8. The **rewrite** statement says that if `Net.Connect()` is called with the HTTP scheme and the data to be transmitted is labeled as secret, this `Net.Connect()` call is replaced by the same call but now with the HTTPS scheme. The specification of this security automaton in EXSPoX is shown in Figure 4.9. The first argument of the **rewrite** statement in the EXSPoX policy set constrains the pointcut: We are only interested in `Net.connect()` calls with the scheme `HTTP` and data labeled as secret. In this case, the call is replaced with a new call according to the second pointcut definition: The same function call, but now with the HTTPS scheme. The EXSPoX policy needs to be finally converted into a concrete Java policy implementation. In case of the HTTPS rewriting, the final Java policy is presented in Figure 4.10.

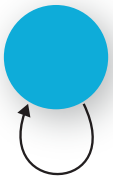
4.6.3 Policy Enforcement

The actual enforcement mechanism of AppGuard policies is based on a novel approach to inline reference monitoring, which was designed and implemented by Philipp von Styp-Rekowsky. In general, AppGuard follows the idea pioneered by Erlingsson and Schneider to rewrite application binaries to directly inline security monitors and checks

```
(edge name="SecretCursor"
    (and
        (cid1:id1=call "Cursor ContentResolver.query(URI,Content)")
        (argval cid1 1 (streq "Contacts"))
        (argval cid1 2 (streq "PhoneNumber"))
        (nodes id1 +)
    )
)
(edge name="MoveToNode"
    (cid2:id2=call "boolean id1.moveToNext())
)
(edge name="SecretPhoneNumber"
    (and
        (cid3:id3=call "String id1.getPhoneNumber()")
        (nodes id3 +)
    )
)
(edge name="MoveToNode2"
    (cid4:id2=call "boolean id1.moveToNext())
)
(edge name="SecretPhoneNumber2"
    (cid5:id3=call "String id1.getPhoneNumber()")
)
(edge name="SecretByte"
    (and
        (cid6:id4=call "byte[] id3.getBytes()")
        (nodes id4 +)
    )
)
(edge name="Declassification"
    (cid7:id5=call "byte[] MessageDigest.digest(id4)")
)
```

Figure 4.7: EXSPoX policy to enforce the secrecy of contact entries

at security relevant method invocations. Based on security and privacy policies, the monitor code either allows or disallows such invocations. In case method invocations are prevented, AppGuard allows the introduction of an alternative control flow, for example, by returning mock values or executing different functions, in order to prevent applications from crashing and to retain functionality as far as possible. The novel approach in AppGuard can be seen as a hybrid approach of caller-side and callee-side rewriting (cf. Figure 4.11 for a comparison of caller-side rewriting, callee-side rewriting, and, finally the new call-diversion approach as implemented in AppGuard). In order to minimize the monitor code that needs to be integrated, it is preferred to inline code directly to the called function. Besides the benefits regarding the amount of code, callee-side rewriting has further the advantage that one cannot miss relevant joinpoints, i.e., it is implied by design that all invocations of a particular function are monitored. Although callee-side rewriting is the preferred approach, it cannot be applied to Android,



rewrite

(call Net.Connect(*scheme,url,data*) && *scheme*==HTTP && issec(*data*))

(call Net.Connect(*scheme,url,data*) && *scheme*==HTTPS)

Figure 4.8: Automata for policy that rewrites HTTP connections to HTTPS connections for secret data.

```
(edge name="EnforceHTTPS"
    (rewrite
        (and
            (cid1:id1=call "Net.Connect(Scheme,URL,Data)")
            (argval cid1 1 (streq http))
            (argval cid2 3 (secret))
        )
        (and
            (cid2:id2 = call "Net.Connect(Scheme,URL,Data)")
            (argval cid2 1 (streq https))
        )
    )
)
```

Figure 4.9: EXSPoX policy to enforce HTTPS connections for data labeled as secret

since all relevant system libraries are sealed. These libraries are part of the Android SDK and belong to the system itself. They can only be inlined if one modifies the firmware or roots the devices, which is exactly what AppGuard does not aims at. AppGuard aims at a solution that can be installed and configured by laymen users on standard consumer devices. Since callee-side rewriting is not possible, the initial approach of AppGuard followed the idea of caller-side rewriting where every relevant method invocation is monitored by wrapping the monitor around the security-relevant function call [P6, P3].

In order to both improve the performance of the inlining process and enable policy updates at runtime without the requirement of rewriting applications again and again, the enforcement mechanism of AppGuard was further improved. A new approach to callee-side rewriting was invented, which mimics the classic behavior of callee-side rewriting when libraries themselves cannot be rewritten [P10, P5, P4]. The idea is to change function pointers in the memory of the Dalvik virtual machine and, thereby, to redirect calls to security relevant functions to monitor functions. The rewriting is achieved by a custom native library, which is integrated to third-party applications by AppGuard. Two important features of the call-diversion approach by AppGuard are its support for monitoring native calls to Java functions as well as its ability to handle Java

```
class HttpsRedirectPolicy extends Policy {
  @MapSignatures({"Ljava/net/URL;->openConnection()"})
  public void checkConnection(URL url) throws Exception {
    if (redirectToHttps(url)) {
      URL httpsUrl = new URL("https", url.getHost(), url.getFile());
      URLConnection returnValue = httpsUrl.openConnection();
      throw new MonitorException(returnValue);
    }
  }
}
```

Figure 4.10: Java policy for redirecting HTTP-based connections to HTTPS-based connections (cf. (P5)).

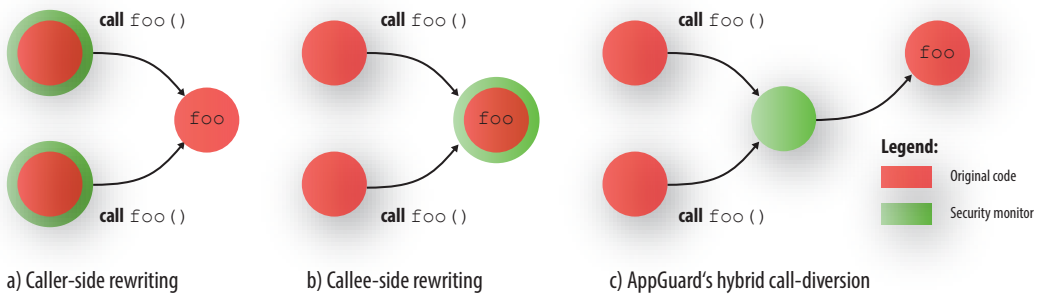


Figure 4.11: Comparison of caller-side and callee-side rewriting with AppGuard's hybrid call-diversion approach.

reflection. The inlining process has proven its robustness with a success rate of 100% in achieving a correct `dex`-file. Moreover, 99.6% of the applications installed from Google's app store *Google Play* [90] were still running stable. A major share of the unstable applications after the inlining process had problems with the exception handling [P4].

4.7 Performance Evaluation

In the following we provide insights on AppGuard's performance. All measurements were either conducted on a Google Galaxy Nexus smartphone with an Texas Instruments OMAP 4460 dual-core CPU of 1.2 GHz and 1GB RAM or on a commodity notebook with an Intel Core i5-2520M dual-core CPU of 2.5 GHz and hyper-threading as well as 8GB RAM. The Galaxy Nexus phone was running its stock Android version 4.1.2.

First, we analyzed the performance of the inlining process (cf. Table 4.3 for an overview of our results.). We compared the inlining time on the Galaxy Nexus smartphone with the inlining time on the commodity notebook and list additionally both the corresponding size of the application package (`APK`-file) and the size of the actual bytecode (`classes.dex`). Since our inlining process directly works on Dalvik bytecode,

Table 4.3: Evaluation of the instrumentation process: Listing sizes of `Apk`- and `Dex`-files and both the inlining time on a commodity desktop and on a smartphone. (cf. (P4))

App (Version)	Size (kB)		Time (sec)	
	APK-file	DEX-file	PC	Phone
Angry Birds (2.0.2)	15.018	994	5.8	39.3
APG (1.0.8)	1.064	1718	0.7	10.1
Barcode Scanner (4.0)	508	352	0.1	2.6
Chess Free (1.55)	2.240	517	0.3	4.2
Dropbox (2.1.1)	3.252	869	0.5	10.2
Endomondo (7.0.2)	3.263	1635	0.7	16.6
Facebook (1.8.3)	4.013	2695	1.2	26.4
Instagram (1.0.3)	12.901	3292	3.0	44.3
Post mobil (1.3.1)	858	1015	0.2	5.8
Shazam (3.9.0)	3.904	2642	1.2	26.1
Tiny Flashlight (4.7)	1.287	485	0.1	2.9
Twitter (3.0.1)	2.218	764	0.3	8.9
Wetter.com (1.3.1)	4.296	958	0.4	10.7
WhatsApp (2.7.3581)	5.155	3182	0.8	27.7
Yuilop (1.4.2)	4.879	1615	0.8	19.7

we save time in comparison to other approaches (for example, [22, 181]), which need to convert the Dalvik bytecode to Java bytecode and vice versa. Since we perform our novel hybrid inlining approach, AppGuard profits additionally from the fact that we can save the time both for finding caller-side invocations and for inlining the monitor code at each position where the security relevant method is called. AppGuard combines all monitoring logic into a single library that can be integrated as is and the actual inlining effort is reduced to loading are monitor library at application startup. So the inlining time itself is minimal. The majority of the overhead introduced by the inlining process stems from repackaging the original `APK`-file. Overall, as can be seen in Table 4.3, the inlining times on the notebook are much faster than on the smartphone; nevertheless, the inlining times on smartphones are among the fastest of all on-the-phone inlining approaches we are aware of. Moreover, user feedback indicates that the achieved overhead is already reasonable for a production ready application: users are willing to accept such a overhead during the installation process if they can achieve stronger security and privacy guarantees. Regarding the overhead in size introduced by our inlining process, AppGuard adds the call-interposition and monitoring library as native code, which adds in total about 3.7 kB to each application.

Besides the measurements regarding the inlining process itself, we additionally analyzed the runtime overhead introduced by AppGuard. We performed microbenchmarks for three different function calls with a different runtime complexity (`Socket-><init>()`, `ContentResolver->query()`, and `Camera->open()`) to achieve a good overview on AppGuard's impact on the runtime performance. We measured the execution time of the original function call and the guarded call based on `System->nanoTime()` and invoked

Table 4.4: Evaluation of the runtime: microbenchmarks for original function calls in comparison to guarded function calls with policies disabled (cf. (P4)).

Function Call	Original Call	Guarded Call	Overhead
`Socket-><init>()`	0.0186 ms	0.0212 ms	21.4%
`ContentResolver->query()`	19.5229 ms	19.4987 ms	0.8%
`Camera->open()`	74.498 ms	79.476 ms	6.4%

before each measurement cycle a garbage collection to minimize noise in our measurements as well as possible. The original function call was evaluated without any code being inlined, whereas the guarded call was executed while no policy was enforced. The latter case means that the monitor code was executed, but the execution of the respective function was permitted. We decided to measure this configuration of the guarded call in comparison to our baseline measurement, as it comes closest to the original execution. If policies are enabled, the execution might actually be faster than the original call because the policy does might not allow the execution of the measured function and simply forces it to return. Table 4.4 lists the median execution times for all functions and measurements. We averaged for each function over ten execution cycles, where `Socket-><init>()` was executed 1000 times per cycle, `ContentResolver->query()` was executed 500 times per cycle, and, finally, `Camera->open()` was executed 25 times per cycle. The overhead, especially for the `ContentResolver->query()`, is really small and we faced during our evaluation measurements in which the guarded calls were actually faster than the original ones. We expect that this was due to noise stemming from the operating system itself, background applications that could not be stopped for the evaluation, and the operating system's task scheduling. In general, we would like to emphasize that the listed numbers are to be considered as a worst-case scenario. The overall runtime impact is only that high if all function calls of an application are guarded. So far we have not recognized any negative impact on the responsiveness of applications inlined by AppGuard or on the overall system usage at all.

4.8 Case Study on AppGuard Policies

The design goal of AppGuard was to put users back in control of installed third-party applications. Users should be able to enforce their own security and privacy requirements on third-party applications, instead of letting the applications dictate which access they get granted. In this section, we provide an overview of a variety of use-cases where AppGuard is highly valuable to both improve system security and enhance the user's privacy. In all cases, we exemplify the utility of AppGuard on real-world applications[11] that where installed from Google Play [90] on smartphones with an unmodified stock Android version. We would like to emphasize that AppGuard was also only installed as

[11] As a disclaimer we would like to emphasize that the applications used were merely chosen to exemplify the utility of AppGuard. They are not considered as malware, nor should our evaluation lead to the conclusion that using one of the applications is dangerous or not recommended. In case we use AppGuard to fix a vulnerability, we explicitly state this.

a stand-alone application. We neither modified the operating system nor provided root privileges, even not temporarily.

During our evaluation of use-cases, we focus, in particular, on (1) the dynamic revocation of Android permissions, (2) the enforcement of new, fine-grained permissions on third-party applications, (3) the enforcement of complex stateful policies, (4) the enforcement of corporate or company policies as part of mobile device management, (5) policy-based quick fixes for vulnerabilities in third-party applications, and (6) the policy-based mitigation of operating system vulnerabilities. In addition, we discuss the potential of using AppGuard to implement a policy-based parental control system.

4.8.1 Dynamic Permission Revocation

The Android permission system does not provide users with the possibility of revoking particular permissions for third-party applications, neither at install time, nor at runtime. Users can either accept the list of permissions requested by a third-party application during the installation process, or they cannot install the application at all. A common example for a permission that requires the possibility to revoke it is the Internet permission `android.permission.INTERNET`. Many applications request this permission without having an obvious reason to do so. Although it might exist a valid justification for requesting it, such as a business model based on in-app advertisements, the decision whether granting it is OK or not should be up to the user: the Internet permission allows applications to send and receive arbitrary data to and from the Internet and there is no way for users to further configure the permission. They even cannot limit the Internet access to a list of dedicated trustworthy servers. AppGuard overcomes this unsatisfactory situation by putting users back into control. Our system allows users to safely revoke permissions at any time. Permissions can be revoked without leading to application crashes since AppGuard provides suitable dummy return values instead of just blocking function calls.

Use-case: Twitter (Version 3.0.1)
In February 2012, the Twitter app[12] was heavily criticized for silently uploading the user's address book (which commonly includes personal information such as phone numbers and e-mail addresses) to Twitter servers [193]. Although Twitter replied that this is part of the find friends feature of the Twitter app, which uses the contact information from the local address book to check whether one of your friends is already on Twitter, they also acknowledged, according to Los Angeles Times, that this information is kept at their servers for 18 month [193]. As reaction to the public criticism on this issue, Twitter updated the app and requests now the explicit consent of the user.

In order to access the address book for the find friends feature, the Twitter app requests the `android.permission.READ_CONTACTS`. The AppGuard `ContactsPolicy` allows users to revoke this permission by default and to grant access to the address book dynamically, which enables users to prevent the Twitter app from uploading the address book. After an application has been installed, users can deny access before the first application start-up. To prevent the Twitter application from crashing, AppGuard is able return empty or fake address book entries if access should not be granted. One

[12] Available at: `https://play.google.com/store/apps/details?id=com.twitter.android`

could even think of implementing a solution where only a special group of contacts can be accessed and processed by an app. The ContactsPolicy of AppGuard monitors queries to the ContentResolver[13] object of Android, which is commonly used to access so-called content providers[14] in Android. Content providers allow to access structured data via a common API. Typical examples besides the ContactsContract provider for all contacts are the CalendarContract for all calendars or the MediaStore provider for accessing audio and video files or images.

Similar to Twitter but a few weeks earlier, also the Path [165] and Hipster app had already been publicly criticized for transmitting address book entries to their servers [239]. But in contrast to Twitter, it is unclear whether the data was later kept at their servers, or whether it was only used for a find friends feature. Nevertheless, the general access to the address book with the later transmission of entries to servers could be blocked by AppGuard the same way as for the Twitter app.

Use-case: Tiny Flashlight (Version 4.7)
The dynamic revocation of Android permissions can further be demonstrated at the example of the Tiny Flashlight app[15]. Its core functionality is, as the name already indicates, to provide a flashlight. This is either accomplished by using the flashlight of the camera, or by turning the whole display white in case no flashlight hardware as part of a camera is available. Since the application requests both the Internet permission android.permission.INTERNET and the permission to use the camera android.permission.CAMERA, this might rise the user's suspicion: it is basically everything required to achieve a perfect bugging device, which would be really difficult to detect for the average user. Moreover, people will usually not understand at first glance, why the camera permission android.permission.CAMERA is required at all because the Tiny Flashlight app also requests the permission to control the flashlight (android.permission.FLASHLIGHT). The reason for requesting the camera permission lies in the different possibilities to access the flashlight of a camera. Depending on the actual hardware, the flashlight can be either accessed directly, or only via the camera interface. Our analysis of the Tiny Flashlight app regarding the Internet permission android.permission.INTERNET indicates that this permission is only required to show in-app advertisements. So, the permission requests of the Tiny Flashlight app seem to have a valid justification.

Since the average user is usually not aware of these details, AppGuard allows users to safely revoke the Internet and the camera permission. Revoking the Internet permission android.permission.INTERNET has basically the effect of an ad blocker. The responsible InternetPolicy monitors all functions of the Android API that can be used to initiate network connections, such as, for example, constructor calls of the Socket classes or the Java.net.url.openConnection() method. Access to the camera is blocked by the CameraPolicy. It monitors access to the android.hardware.Camera.open() method and simulates hardware without a back-facing camera (returns null)[16], since the back-facing camera commonly includes the hardware flashlight. In case of the Tiny Flashlight

[13]Details: http://developer.android.com/reference/android/content/ContentResolver.html
[14]Details: http://developer.android.com/guide/topics/providers/content-providers.html
[15]Available at: https://play.google.com/store/apps/details?id=com.devuni.flashlight
[16]Details: http://developer.android.com/reference/android/hardware/Camera.html

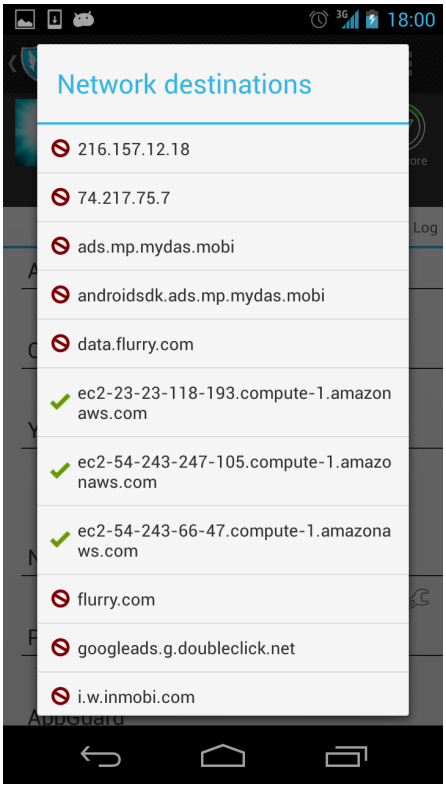


Figure 4.12: Screenshot of the commercial SRT AppGuard app showing the whitelisting of network connections.

app, the `CameraPolicy` can even be enforced without loosing functionality: The app is implemented to also function on devices without a hardware flashlight. In this case, the Tiny Flashlight app simply turns the display white.

4.8.2 Fine-grained Permission Enforcement

Besides the dynamic revocation of Android permissions, AppGuard also allows to define completely new permissions. The new permissions can be both used to restrict functionality that is not yet handled by the current permission system of Android, and to restrict functionality in another way – and most likely more fine-grained – than done by the current Android permissions. A typical example of a more fine-grained enforcement of permissions are our extensions to the `InternetPolicy`. The `InternetPolicy` implements a classical whitelisting approach (cf. Figure 4.12 for the UI of the whitelist in the commercial AppGuard implementation by BackesSRT) where users can decide which server connections they want to explicitly allow. All other connections are subsequently denied by AppGuard. This is in line with the typical expectations of users that an app communicates only with a limited set of servers: for example, the Facebook app should only communicate with Facebook servers.

Use-case: wetter.com App (Version 1.3.1)
The wetter.com app[17] provides users with the latest weather information for user-selected regions. In order to present the latest weather information to the user, the app connects to the servers of wetter.com and updates the current weather forecast. There is simply no reason to connect to other servers for fulfilling the purpose of the app. It only needs to contact the wetter.com servers.

AppGuard allows to enforce a fine-grained configuration of the Internet access by implementing a consequent whitelisting within the `InternetPolicy`. This is achieved by extending the monitoring of all Android API functions that can be used to initiate network connections with an explicit whitelist. Thereby, our monitor is able to only permit connections to destination servers that are on that list. Our implementation of the `InternetPolicy` for wetter.com requires that all allowed servers match against the following regular expression: `^(.+.)?wetter.com$` [P5].

4.8.3 Stateful Policies

Another class of policies requires state information from previous executions. This could be, for example, a policy that prohibits an app to send data to the Internet, if the app accessed before sensitive information like contacts or calendar. Another possibility is to use state information such as a counter to limit the amount of certain actions: Without further confirmation by the user, an app can be, for example, limited by our `CostPolicy` to at most three text messages to premium rate phone numbers.

Use-case: Post mobil (Version 1.3.1) A nice example to demonstrate the effectiveness of the AppGuard `CostPolicy` is the Post mobil app[18], which is the official app by the German postal service. The app allows users as one out of several features to buy stamps online by sending text messages to a premium rate phone number. The stamp is delivered in turn as a digit code of three rows with 4 digits each. According to this procedure, the app obviously also requests the corresponding permission to send text messages (`android.permission.SEND_SMS`). Since granting the permission to send text messages to unknown apps is a potential financial risk for the user, our policy allows to limit the amount of text messages an app is allowed to send. This is achieved by monitoring all relevant Android API functions such as the `SmsManager.sendTextMessage()` and the `SmsManager.sendMultipartTextMessage()` of the `SmsManager` object[19].

4.8.4 Corporate and Company Policies

Many companies leverage mobile device management solutions to integrate mobile devices into their corporate or company networks. However, commonly they can only configure and enforce what the management API of a particular device supports. On Android, this API does not include any possibility to enforce particular policies inside of third party applications. With AppGuard, companies get the chance to enforce corporate or company policies inside of third party applications such as password complexity policies or access prevention to certain resources.

[17] Available at: https://play.google.com/store/apps/details?id=com.wetter.androidclient
[18] Available at: https://play.google.com/store/apps/details?id=de.deutschepost.postmobil
[19] http://developer.android.com/reference/android/telephony/SmsManager.html

Use-case: Enforce Password Policy in APG App (Version 1.0.8)
We exemplify the utility of such policies by instrumenting the APG app[20], a popular implementation of OpenPGP for Android. OpenPGP allows users to encrypt and decrypt data such as e-mails or files based on public key cryptography. Additionally, the app allows to sign data and to subsequently verify the signatures. In order to protect the secret keys used for decryption and signing, the APG app requests users to enter a password during the creation process of the keys. Although APG does not allow users to choose an empty password, it does not enforce a particular password complexity either. Since it is common use in companies to enforce policies about both length and complexity (character set: small and capital letters, numbers, special characters) of passwords, it is also necessary to enforce these policies on mobile devices. While mobile device management solutions are able to use the management API of mobile devices to enforce, for example, a certain complexity for lock screen passwords, such an enforcement is impossible inside of third-party applications. AppGuard overcomes this deficiency and allows administrators to define so-called joinpoints inside of apps that are also monitored by our system. That way, we monitor the procedures that reads in the passphrase of the key and allow the app only to proceed if the password entered during the key creation process matches our policy. In a similar way, AppGuard would also allow companies to enforce a certain key size for the created keys or a particular algorithm.

4.8.5 Quick Fixes for Vulnerabilities in Third-Party Apps

Besides the previously introduced use-cases where the enforced policies mostly aimed at giving users and companies back the control over the privacy and security settings for third-party applications, we introduce in the following two policies that are of general interest. In both cases, the enforced policies aim at mitigating vulnerabilities in applications and the operating system. First of all, we will have a look at how AppGuard can be used to fix vulnerabilities in third party applications. Over and over again it happens that applications make use of HTTP connections for privacy sensitive data. Although most apps use the HTTP over TLS (HTTPS) protocol for login procedures on Web servers, there are still applications that fall back to HTTP right after the authentication. The switch back to HTTP leads to the bad situation that the authentication token required at the server-side to identify users throughout a session[21] is also transmitted in the clear. Attackers are now able to piggyback on the cleartext connection and to steal confidential information such as the authentication token, which allows the attacker to impersonate the current user [125]. In case the Web server supports HTTPS throughout the full session, AppGuard can easily solve the problem using our `HTTPSEverywhere` policy: our system enforces the usage of HTTPS connections for all sensitive data, which successfully protects also the authentication data. Depending on the code executed inside of an app, this can either be achieved by replacing, for

[20] Available at: `https://play.google.com/store/apps/details?id=org.thialfihar.android.apg`
[21] The HTTP protocol itself is by default stateless and requires applications to additionally make use of authentication tokens to link authenticated users to a session.

example, the `HttpURLConnection`[22] object by a `HttpsURLConnection` object[23], or by directly returning the data after establishing the new HTTPS connection. In a similar way, AppGuard could also be used to fix missing or vulnerable implementations of the certificate check within SSL connections. Problems with the certificate check can lead to similar problems as when falling back from HTTPS to HTTP, since one can trick apps into connecting to a wrong, malicious server. The Fraunhofer Institute for Secure Information Technology showed in the end of 2013 that problems with the certificate check still occur. They reported about several popular Android applications with security issues related to such a certificate check [140]. By using AppGuard, also these vulnerabilities could have been mitigated until the fixed applications were available.

Use-case: Endomondo Sports Tracker App (Version 7.0.2)
An application that was prone to leaking the authentication token after a successful login is the Endomondo Sports Tracker[24]. The application uses by default the HTTPS protocol for the login procedure. However, after a successful login it falls back to HTTP and transmits the authentication token for the session in clear such that attackers could easily piggyback on the connection. In order to fix this issue, we implemented and tested our `HTTPSEverywhere` policy that ensures the usage of HTTPS throughout the full session. As previously described, the policy monitors the corresponding functions to open HTTP connections (for example, `Java.net.url.openConnection()`), and either returns a HTTPS connection when a HTTP connection was requested, or it returns directly the data from the HTTPS redirection.

4.8.6 Mitigation of OS Vulnerabilities

AppGuard can also be used to mitigate the impact of vulnerabilities in the Android operating system. Whenever such a vulnerability can be exploited by a third-party application through Android API calls, AppGuard allows to enforce system policies that prevent the particular calls necessary to exploit such a vulnerability. Since our system policies are of course only active in all instrumented applications, the protection can also only achieve an optimal coverage if all third-party applications are instrumented by AppGuard. Similarly, AppGuard can be used to introduce new permissions for yet accidentally unprotected API functions that might harm the user's security or privacy needs.

Use-case: Add Missing Permission to Protect the Photo Storage
That a system like Android might indeed be missing some permissions to protect sensitive APIs was demonstrated by a proof-of-concept exploit named *EvilTeaTimer* app[25] [91] in 2012. At that point of time, Android was missing a protection for the central photo storage where, for example, all photos from the default camera app are stored. Without any protection by a permission that needs to be explicitly requested by third-party applications, every third-party application installed on the Android device can access this highly personal information [52]. If a third-party application additionally

[22] Details: http://developer.android.com/reference/java/net/HttpURLConnection.html
[23] Details: http://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html
[24] Available at: https://play.google.com/store/apps/details?id=com.endomondo.android
[25] Available at: https://github.com/ralphleon/EvilTeaTimer

requests the Internet permission, it can simply upload all images from the photo storage to arbitrary servers on the Internet. AppGuard implements a `MediaPolicy` that prevents instrumented applications by default from accessing the photo storage. User's can use the policy configuration of AppGuard to subsequently grant this permission to individual apps. The policy monitors queries to the `ContentResolver` object that targets at `MediaStore` and prohibits access for unauthorized applications. Since AppGuard is only able to instrument third-party applications[26], the AppGuard-based protection cannot be achieved for applications that were integrated into the operating system by Google or the device vendors (for example, the Google apps such as Google Maps). However, when using an Android device from a particular vendor, users need to trust both Google and the vendor anyway.

Use-case: Prevent Intent-based Cross-site Scripting in Android Browser
Another vulnerability that can easily be defeated by AppGuard is the intent-based cross-site scripting attack on the Android browser as it was presented by Backes et al. [19] in 2011 for Android version 2.3.4. intents[27] are special messages in Android that describe which action (for example, ACTION_VIEW, ACTION_CALL, or ACTION_EDIT) the message receiver should perform (for example, the telephony app or the browser app) based on the data that is included in the message [5]. Applications can register as message receivers for specific actions (for example, ACTION_VIEW) and data types by using so-called intent filters[28], where the data type[29] of an intent filter can be specified based on the scheme used (for example, HTTP), a host (for example, `google.com`), a port (for example, `80`), a particular path or a path prefix/pattern, and finally a MIME type (for example, `image/jpeg` or `text/plain`). The system decides based on the list of registered intent filters to which component an intent is actually delivered. In case several applications register for the same type, either a default can be set, or the user is asked to choose the destination app every time an intent with the particular action and data combination is send.

If a third-party application invokes now an intent with the action ACTION_VIEW and includes as URI a Web address (for example, `https://play.google.com`), this intent will most likely be received by the default Web browser on the system: the Android browser (for Android version 2.3.4). The Android browser now takes the included URL from the intent and opens it in a new browser tab. Although we consider this behavior by itself already as a vulnerability, since also applications not holding the Internet permission can send out such intents, which fully suffices to upload data to the Internet, the attack presented by Backes et al. goes even a step further: the third-party application sends afterwards a second intent with the same action, but now with a JavaScript scheme. Due to a bug in the Android browser, this JavaScript code is not executed in a new browser tab, but in context of the previously opened tab which loaded `https://play.google.com`. A malicious application could now, for example, use the combination of these two intents to steal session tokens or cookies in order to impersonate

[26] Instrumenting system applications would require to modify the OS image or at least root privileges, which cannot be achieved for laymen users and without voiding the user's warranty.
[27] Details: `http://developer.android.com/reference/android/content/Intent.html`
[28] Details: `http://developer.android.com/guide/topics/manifest/intent-filter-element.html`
[29] Details: `http://developer.android.com/guide/topics/manifest/data-element.html`

the user or to upload the session token/cookie to external servers. AppGuard is able to both prevent the intent-based Internet access for applications not holding the Internet permission and the cross-site scripting attack based on the combination of these two particular intents. It simply monitors the functions used for invoking intents and ensures that particular combinations of sender, receiver, and intent content are prevented.

4.8.7 Parental Control

As a final use-case for enforcing AppGuard policies, we would like to discuss the usage of AppGuard for parental control. As a straight forward way, one could, for example, simply enforce a policy that requests from the user for all applications not suitable for children a special password. Of course, this is only effective if AppGuard instruments really all applications. Another possibility would be to employ a whitelist for all suitable/child-friendly Web sites that are enforced on all instrumented applications by AppGuard. One could even think of time limits for child use such as at most 30 minutes per day. In principle, AppGuard is able to enforce these kinds of policies and fulfills the needs for parental control. It even allows restricting content within third-party applications without requiring the developers of such apps to offer direct support for parental control. Nevertheless, AppGuard has also one downside in this scenario: Its effectiveness and protection depends on the ability to achieve a 100% coverage for installed applications; independent of the fact whether an app was installed by Google or the vendor, or another third-party. For system applications from Google, or applications from the phone's/tablet computer's vendor, the instrumentation by AppGuard requires, in contrast to the instrumentation of third-party applications, root privileges. These system applications or the applications from the vendor are the only reason why AppGuard cannot achieve the desired coverage on Android.

4.9 Related Work

The permission system of the Android operating system has been subject of many scientific evaluations [21, 172, 84] and many improvements or alternatives have already been proposed (e.g., [160, 56, P5]). It was heavily criticized in recent years, mostly because the system is inflexible and does not allow users to configure their needs with respect to security and privacy, but also because the coverage of single permissions is not well documented [172, 84] and some API functions are not protected at all [91, 19]. In 2010, Barrera et al. conducted an empirical analysis of the Android permission system based on 1,100 Android applications and so-called self-organizing maps [21]. Their results show that although Android provided at that point in time about 110 permissions, only a few of them were really frequently used by developers. As expected, the permission used most is the Internet permission: it is used by more than 60% of the evaluated applications. Barrera et al. discuss in their results already the trade-off between getting more fine-grained permissions to achieve the user's needs and the problem for developers to understand which permissions are actually needed. They think that this might lead to a situation where developers for reasons of simplicity simply request more permissions than actually needed to be on the safe side.

Porter Felt et al. analyze the effectiveness of application permissions at the example of Google Chrome and at the example of Android [171]. Both share a similar concept for permissions as they both request them at install-time without offering users any further choice than not installing applications with undesired permissions. Their results indicate that offering application-based permissions is indeed an advantage in comparison to running all applications with full user rights (as done on classic desktop operating systems). Most of the time only a small subset of permissions is requested, which reduces the potential impact of in-app vulnerabilities. But users might get used to accepting even more dangerous permissions such as the Internet permission in Android, simply because so many apps need it. This might in the end lead to a less thoughtful decision in comparison to cases where warnings are really exceptions and raise the user's attention.

Both Porter Felt et al. [172] and Gibler et al. [84] provide an in-depth analysis of the Android source code to figure out which API functions are actually affected by permissions. Porter Felt et al. create a permission map (which API functions require which permissions) and introduce with *Stowaway* a static analysis tool that first identifies all Android API calls in an application and uses afterwards the permission map to detect over-privileged applications (i.e., applications requesting more permissions than their code actually needs) [172]. The permission map of Porter Felt et al. lists 1,259 API calls that require permissions, which is surprising, since they found only 78 API calls in the Android documentation that listed the requirement of a permission. Similar to Porter Felt et al., Gibler et al. create a permission map for Android and show how this can be used in combination with static analysis to guide developers to prevent over-privileged applications by proposing the minimal set of required permissions [84]. But Gibler et al. go even one step beyond: they used their system to analyze apps regarding potential privacy leaks and found 3,258 of 23,838 applications that potentially leak system or user-unique data such as the phone identifier, location or wireless network information, or even recordings from the microphone. Moreover, the authors analyze potential data leaks in ad libraries. Vidas et al. manually parse the API documentation to create a permission map that is subsequently used within a plug-in for the Eclipse IDE to assist developers of apps in fulfilling the principle of least privilege [229]. In their case study on *Building Security into "Off-The-Shelf" Smartphones*, Stavrou and colleagues from NIST speculate about the reasons for developers to request more permissions than actually required: they name the lack of understanding of the existing permission system, simply the developers' convenience, or that developers might expect to add functionality requiring the additional permission in the future [212]. Stavrou et al. state to have detected thousands of Android applications that request more permissions than the app's functionality actually requires.

The evaluations of the Android permission system have shown that the current state of this system does not provide users with the flexibility to configure OS-enforced security and privacy settings of third-party applications according to their needs. Even worse, missing permissions open attack vectors and put the user's data at risk. Research has proposed several systems that address the previously described issues with the permission system and we will now first discuss the positive and negative aspects of the systems closely related to AppGuard; i.e., systems based on inline reference monitoring

Table 4.5: Comparison of Android IRM approaches (cf. (P4))

Feature	1	2	3	4	5	6	7	8	Runtime Overhead
Dr. Android [115]	✓	–	E	◐	◐	–	–	–	10-50%
Aurasium [242]	✓	–	I	◐	◐	–	–	–	14-35%
I-ARM-Droid [60]	✓	–	I	–	◐	–	–	✓	16%
URANOS [197]	✓	✓	I	–	✓	–	–	✓	Not measured
Barthel et al. [22]	✓	✓	E	–	–	–	–	✓	Not measured
DroidForce [181]	✓	–	E	–		✓	–	✓	Not measured
AppGuard [P6, P5, P10, P3, P4]	✓	✓	I	◐	✓	✓	✓	✓	1-21%

Legend: 1. No Firmware Mod. 2. On Phone Instr./Updates 3. Monitor
4. Native Methods 5. Reflection 6. Policy Lang. 7. Data Secrecy
8. Parametric Joinpoints; ✓: full support, ◐: partial support

and systems that rewrite third-party applications and facilitate an external reference monitor such that they can be deployed to standard consumer devices without requiring modifications of the firmware, root privileges or the like. Afterwards, we will discuss further, less-closely related work on security and privacy topics for Android.

4.9.1 Inline Reference Monitoring

Our discussion on approaches to inline reference monitoring is clustered into two areas: First we will discuss approaches targeting the Android operating system, and afterwards related inline reference monitoring approaches in general.

4.9.1.1 IRM on Android

A comparison of all IRM approaches for Android is provided in Table 4.5. We compare the approaches based on the following features: (1) the possibility to deploy the system without requiring modifications to the firmware of a mobile device, root privileges, or the like; (2) the support for an on-the-phone instrumentation and updates; (3) the usage of an (E)xternal or an (I)nternal reference monitor[30]; (4) the protection against native methods; (5) the support for handling reflective calls; (6) the integration of a policy language; (7) the support for data secrecy policies to separate secrets in third party applications; the flexibility in the selection of joinpoints; and, finally, (9) the introduced runtime overhead. Although AppGuard is the only approach that supports all of the desired features, we would like to emphasize that Dr. Android, I-ARM-Droid, and Aurasium constitute concurrent work of AppGuard, whereas URANOS, DroidForce, and the paper by Barthel et al. were published later.

Dr. Android and Mr. Hide by Jeon et al. makes use of an external reference monitor that is installed as stand-alone Android service, the so-called *Mr. Hide service* [115].

[30] Although the policy decision logic with the reference monitor might be deployed as an external application, we still consider these approaches as some sort of inline reference monitoring: The platform does not support an integration into the operating system so that the actual checks that refer to the external reference monitor still need to be inlined to the individual third-party applications.

The core idea of the approach is to remove all permissions requested by a third-party application in its manifest file (`AndroidManifest.xml`) and to proxy all the privileged API calls through the Mr. Hide service. That way, the Mr. Hide service is able to enforce application and system specific policies when third-party applications attempt to use such privileged calls. However, this is limited to security critical functions that are actually proxied through Mr. Hide. The system is not designed to monitor arbitrary Java functions for policies targeting at application specific code, since these calls would not go through the Mr. Hide service or at least would require to be individually integrated. Removing all permissions of third-party applications is clearly beneficial for the impact of vulnerabilities in monitored third-party applications; however, the Mr. Hide service requires now all permissions, which makes it a prominent target for attacks. Since Mr. Hide is not able to drop permissions dynamically[31], the service cannot follow the principle of least privilege. To mitigate this, one could consider implementing dedicated services for every single application such that the proxies also only hold the permissions required by its corresponding app. The integration of the system into third-party applications is achieved by integrating an extra `dex`-file with the `hidelib` into each application. It encapsulates the entire protected API in a new namespace and handles the inter-process communication with the Dr. Hide service transparently. A nice fact of the external monitor is that the approach automatically prevents privileged native calls of the application, even if the low-level system functionality is re-implemented. The third-party application is simply missing the required permissions. Moreover, the approach also covers reflective calls to security critical functions, as long as the resolved function is later monitored by Mr. Hide. A limitation of the approach in comparison to AppGuard is that it does not support the instrumentation of apps on the phone or automatic updates; this needs to be done on a desktop computer or potentially via a Web service that directly provides the instrumented application. Moreover, the overhead of the system by Jeon et al. is, due to the external service and the required inter-process communication, quite high: it has the highest overhead of all approaches we compared[32]. The *Dr. Android and Mr. Hide* system might also be prone to timing issues since the application needs to synchronize with the Dr. Hide service at startup. Without having this service up, a monitored application might not be able to start. AppGuard has here clearly an advantage: The inlined monitor makes it much faster and the decision logic is locally integrated into the applications. Only AppGuard's policy configuration is provided from the AppGuard application, but such way that it can always be read from the file system. Regarding the policies, *Dr. Android and Mr. Hide* does not integrate a high-level policy language or support for the code-based separation of secrets. Finally, *Dr. Android and Mr. Hide* does not allow the parametric specification of joinpoints.

I-ARM-Droid [60] is a system by Davis et al. that is, similar to AppGuard, based on inline reference monitoring, i.e., the security monitor is directly embedded into the third-party applications to be monitored. Similar to our initial instrumentation approach for AppGuard, *I-ARM-Droid* follows the idea of caller-site rewriting and instruments all invocations of security relevant API methods to enforce custom security policies. The system supports parametric joinpoints and their idea of rewriting security

[31] This restriction is given by the Android permission system.
[32] Out of all approaches that actually measured the overhead.

relevant API methods is based on redirecting the original API calls to so-called proxy methods. The proxy method decides based on the original arguments and the active policy either to permit and execute the original call, or to change the control flow and perform an alternative action/stop the execution. Using caller-side rewriting has the disadvantage that each invocation of a function needs to be found and instrumented, which includes not only the risk to miss one execution (for example, due to reflective calls), but also introduces quite some overhead for the instrumentation process. But since Androids system libraries are sealed, classical callee-side instrumentation is not feasible. With AppGuard we overcome this unsatisfactory situation and perform a novel rewriting technique at runtime in the memory of the Dalvik virtual machine, which is kind of a hybrid caller-site/callee-site solution. But this provides us with all the advantages of a callee-site rewriting. In contrast to AppGuard, *I-ARM-Droid* does not support the rewriting of applications on the phone, which requires again either a desktop computer or an additional Web service. Although *I-ARM-Droid* is able to detect reflective and native calls, both are not handled in depth. For native code no further action is taken and for reflective calls *I-ARM-Droid* currently simply prevents their execution without providing special logic to handle them. Regarding our desired features, *I-ARM-Droid* also neither includes a high-level policy language nor does it support the separation of secrets as done by AppGuard. Performance-wise, *I-ARM-Droid* produces, as expected, less runtime overhead as *Dr. Android and Mr. Hide*. *I-ARM-Droid* [60] is a system by Davis et al. that is, similar to AppGuard, based on inline reference monitoring, i.e., the security monitor is directly embedded into the to be monitored third-party applications. Similar to our initial instrumentation approach for AppGuard, *I-ARM-Droid* follows the idea of caller-site rewriting and instruments all invocations of security relevant API methods to enforce custom security policies. Their system supports parametric joinpoints and their idea of rewriting security relevant API methods is based on redirecting the original API calls to so-called proxy methods. The proxy method decides based on the original arguments and the active policy either to permit and execute the original call, or to change the control flow and perform an alternative action/stop the execution. Using caller-side rewriting has the disadvantage that each invocation of a function needs to be found and instrumented, which includes not only the risk to miss one execution (for example, due to reflective calls), but also introduces quite some overhead to the instrumentation process. However, since Android's system libraries are sealed, classical callee-side instrumentation is not feasible. With AppGuard we overcome this unsatisfactory situation and perform a novel rewriting technique at runtime in the memory of the Dalvik virtual machine, which is kind of a hybrid caller-site/callee-site solution. But this provides us with all the advantages of a callee-site rewriting. In contrast to AppGuard, *I-ARM-Droid* does not support the rewriting of applications on the phone, which requires again either a desktop computer or an additional Web service. Although *I-ARM-Droid* is able to detect reflective and native calls, both are not handled in depth. For native code no further action is taken and for reflective calls *I-ARM-Droid* currently simply prevents their execution without providing special logic to handle them. Regarding our desired features, *I-ARM-Droid* also neither includes a high-level policy language nor does it support the separation of secrets as done by AppGuard. Performance-wise, *I-ARM-Droid* produces, as expected,

less runtime overhead as *Dr. Android and Mr. Hide*.

Completely different to *Dr. Android and Mr. Hide* and *I-ARM-Droid* is *Aurasium* [242], an approach introduced by Xu et al. Instead of intercepting Java function calls by inlining a reference monitor to the Dalvik or Java byte code, *Aurasium* directly intercepts the interaction between third-party applications and the operating system. In case a third-party application wants to access permission protected system functions in Android (for example, as part of `libc`), this is either achieved by directly accessing the resources (for example, for the Internet connection) or by accessing them via system services such as the *Location Manager Service* for GPS access. In both cases, this leads in the end to system calls; however, only the first case with the direct access to functions can be efficiently monitored by *Aurasium*. In the second case, the final system call is executed by the system service and, thereby, outside of the application context *Aurasium* is able to access. In order to also cover this second case where the access protection is handled inside a system service, *Aurasium* needs to monitor the `Binder`-based inter-process communication of the third-party application with the system service. But this is quite challenging: *Aurasium* needs to identify the target application and needs to re-establish the context of a particular `Binder`-communication. This includes both to reconstruct parameters of the inter-process communication and high-level Java objects by unmarshalling byte-streams in native code [242]. Without this information, it is impossible for *Aurasium* to evaluate its security and privacy policies. Besides the effort required to reconstruct the high-level information, another weak spot of *Aurasium's* reconstruction is the fact, that it might break with every change of the Android operating system[33]. Implementation-wise, the system adds a native library for achieving the call interposition to each application package. The actual call interposition is achieved by rewriting the function pointers of `libc` functions. The policy logic of *Aurasium* is integrated as Java code, since already the authors themselves claim that native code "is generally difficult to write and test" [242]. Another weakness of *Aurasium* is its limitation to only allow the monitoring of permissions protected API calls that end up in system calls. Calls to the Android API that do not involve system calls cannot be monitored. The same holds true for application specific code that does not make use of the Android API. The high-level Java code can only be treated as a black box by *Aurasium*. A typical scenario that cannot be covered by *Aurasium* is, for example, the declassification of secret data: if an app accesses the address book to receive a phone number and hashes this number afterwards, it is no risk to send the hashed phone number to the Internet (for example, as part of a find friends feature in social network applications). In this particular scenario, *Aurasium* is only able to monitor and prevent the access to both the address book and the Internet, but the system cannot detect the declassification. Positive is the support of *Aurasium* for a a centralized security manager to handle policy decisions centrally. This allows *Aurasium* to detect collusion attacks and to enforce device policies based on global state. In contrast to AppGuard, *Aurasium* does not allow an instrumentation on the phone. In principle, the approach can monitor system calls from native code, but only as long as the functionality from system libraries such as `libc` is not re-implemented inside of the native code. AppGuard has here a similar limitation; it can only monitor Java calls from native code. Reflective

[33]The authors mention one change from Android version 2.2 to 3.x that affected their implementation.

calls of *Aurasium* can only be covered as long as the reflective call ends up in a system call. Although this coverage achieves enough protection for invocations of security critical functions, *Aurasium* does not support fine-grained policies that directly target the reflective calls. In comparison to AppGuard, *Aurasium* neither provides a high-level policy language nor does it support the separation of secrets. The overhead of the approach highly depends on the effort to recover all information required to evaluate the policies and is stated with 14-35%.

The *URANOS* system was introduced by Schreckling et al. in 2013 [197] and makes use of inline reference monitoring. Similar to the previously described approaches, the *URANOS* system neither requires to modify the firmware of an Android device, nor does it require root privileges or the like. In contrast to *Dr. Android and Mr. Hide*, *I-ARM-Droid*, and *Aurasium*, the *URANOS* system provides, similar to AppGuard, the full instrumentation toolchain on the Android device and even supports updates for instrumented applications. *URANOS* performs a static analysis on unmodified consumer devices in order to detect when permissions are actually required. Afterwards, it allows removing redundant permissions and enables users to apply their desired permission configurations. The system introduces so-called security wrappers to revoke classical Android permissions. In case permissions required to call a function are removed, *URANOS* provides a mechanism to gracefully handle the revocation of these permissions based on Java exceptions. The exceptions are used to divert control flow to alternative handlers if a particular function call is denied, for example, to return mock values. The authors focus in the paper on the revocation of Android permissions and on enforcing user-desired permission configurations, but its generic implementation of inline reference monitoring seems to also support more fine-grained policies. *URANOS* monitors reflective calls, however, the paper does not mention how native calls are handled. In comparison to AppGuard, *URANOS* neither provides a high-level policy language nor does it support the separation of secrets. Unfortunately, the authors did not include microbenchmarks to measure the runtime-overhead of the instrumented applications. The previously described approaches have shown that this overhead is commonly not negligible.

Barthel et al. introduce a system based on an external reference monitor [22]. The system supports the instrumentation of applications on Android devices based on two different approaches: The first approach is called Soot [209] and entitles itself as a Java optimization framework. It is used in conjunction with Jimple [118], a typed intermediate representation. The second approach used is called ASM [16], a Java bytecode manipulation and analysis framework. Both approaches work on Java bytecode, which means that the Dalvik bytecode is first transformed to Java bytecode by the `dex2jar` tool. This is time consuming and takes already between 10 and 20 seconds for a 250 KiB `classes.dex` file[34]. Furthermore, Soot with Jimple is not really designed to be used on mobile devices, which we can also see in the following approach by Rasthofer et al. called *DroidForce*. With such limited resources on mobile devices, the Soot-based approach results in a quite slow instrumentation procedure. On average, the Soot-based process needs for the overall instrumentation with all bytecode transformations per application about 136.1 seconds on the fastest device used, which is a tablet device

[34]These measurements depend on the devices that were used during the evaluation by Barthel et al.

running Android 4.0.3 with a single core performance[35] of 1.4GHz, 1GB of RAM and a heap size of 48MiB. In comparison, AppGuard requires for the rewriting of large apps such as Instagram[36] with a `classes.dex` file of 3292 kB only 44.3 seconds; for smaller applications like the Tiny Flashlight app[37] with a `classes.dex` file of 485 kB, this drops to 2.9 seconds and AppGuard's main time consumption stems from the repackaging of the application packages (cf. Figure 4.3). In contrast to the Soot-based approach, the ASM-based version of the system by Barthel et al. compares performance-wise much closer to AppGuard, but the overall performance is still quite slow: the whole ASM-based instrumentation approach takes on average 71.4 seconds. A severe limitation of both instrumentation procedures is that even on the fastest device with the highest memory during their evaluation, only 14.6% of all application could be successfully instrumented by the Soot-based procedure, and only 25.3% could successfully be instrumented by the ASM-based procedure. Root cause of these low numbers seem to be problems with the transformation tool `dx` that transforms Java bytecode to Dalvik bytecode. Similar to *URANOS*, Barthel et al. do not provide details on the runtime overhead of their system. In addition, the authors do not discuss native methods or reflection so that we conclude for our evaluation, that both cases are not covered. Similar to the previously discussed approaches related to AppGuard, the system by Barthel et al. neither provides a high-level policy language, nor does it supports the separation of secrets. But according to the description of Barthel et al., the system at least supports parametric joinpoints for the instrumentation.

DroidForce[181] is the most recent approach to dynamic runtime enforcement on Android. It was introduced by Rasthofer et al. in 2014 and is the second solution that relies on Soot with Jimple. Currently, the system does not support an on-the-phone instrumentation of third-party applications, which is most likely due to similar problems as they were faced by Barthel et al. The authors argue that they conduct the instrumentation off-the-phone due to the static analysis of applications. From their description it is unclear whether the pure instrumentation would work on a phone and the previously described result of Barthel et al. [22] indicates that this is not yet possible with the performance of current mobile devices: Barthel et al. state in their observation 6 for Soot that "Only the smallest applications (in terms of Dalvik bytecode) can be converted." *DroidForce* facilitates an external monitor that includes the policy decision logic to enforce system-wide policies, which is also possible for AppGuard with the difference that the central AppGuard application is only used to maintain state. The actual decision logic and enforcement mechanism of AppGuard is integrated into the third-party application within AppGuard's instrumentation process. Similar to *DroidForce*, AppGuard also supports data-centric and dynamic policies. Policy updates of AppGuard do also not lead to a re-instrumentation of an app. A feature of *DroidForce* that is not yet supported by AppGuard, but can be easily implemented, is the support of time constraints. A similar effect as the dynamic tracking of information flows as integrated by *DroidForce* is achieved in AppGuard through our separation of

[35] The tablet has four cores, but the authors state that they do not make use of the multi-core architecture.

[36] Available at: https://play.google.com/store/apps/details?id=com.instagram.android

[37] Available at: https://play.google.com/store/apps/details?id=com.devuni.flashlight

secrets to prevent data leaks and our enforcement of information flow control policies. A static analysis of flows is not part of AppGuard itself, but *Bati* [T1] was designed to provide this input and to subsequently prepare for policies specifically targeted at the individual behavior of third-party applications (cf. Figure 4.3). The authors of *DroidForce* mention that their system does not monitor native calls, but they do do not mention how *DroidForce* handles reflective calls: therefore, we did not check this feature in the feature comparison chart. As *DroidForce* is based on Soot with Jimple, it should, in principle, allow an easy integration. Rasthofer et al. do not perform a detailed analysis of the runtime overhead introduced by single applications, they only state that none of the instrumented applications "exhibited any perceivable slowdown in the user experience". If we look at the *Dr. Android and Mr. Hide* system [115], which also leverages an external monitor in a dedicated application or service, we can see that the Binder-based inter process communication with this external service introduces de facto quite some overhead. Even if this is not directly noticeable by users, the overhead should clearly introduce a noticeable negative impact on the battery runtime. Similar to AppGuard, *DroidForce* ships with a high-level policy specification language and supports parametric joinpoints.

4.9.1.2 IRM for Non-Android Systems

Erlingsson and Schneider introduced *Inlined Reference Monitoring* in 1999 through the development of the *Security Automata SFI*[38] *Implementation (SASI)* system for Intel x86 assembly code and for Java bytecode (JVML) [68]. The concepts behind *Inlined Reference Monitoring* go back to various approaches that targeted at rewriting binary code such as the *ATOM* system for building customized program analysis tools [211] or the *Naccio* system by Evans and Twyman [70] for enforcing safety policies. In a follow-up work Erlingsson and Schneider introduced in 2000 the successor of *SASI*, the *Policy Enforcement Toolkit (PoET)*, which leverages its own *Policy Specification Language (PSLang)* and focuses now completely on JVML applications [67]. Both the SASI and PeET/PSLand system are nicely summarized in Erlingsson's PhD thesis of 2004 [66]. Schneider introduced in his paper on enforceable security policies the *Execution Monitoring (EM)* class of policies that can be enforced by monitoring execution steps of systems [194] as well as corresponding automata to describe these policies. Although the focus is on terminating target applications if an enforced policy is violated, the paper also discusses the possibility to respond on violations by substituting the critical execution step by an uncritical execution step. The class of automata capable of transforming the execution of a program without forcing it to stop in case of policy violations was later formally introduced as *Edit Automata* by Bauer, Ligatti, and Walker [24, 132]. Based on *Edit Automata*, Bauer et al. introduced in 2005 the *Polymer* system, which enables the specification and enforcement of complex runtime security policies for Java applications [25]. The system is designed to facilitate the composition of policies and uses its own policy language. Chen and Roşu introduce with *Java-MOP* a software development and analysis framework for Java that enables the runtime monitoring of programs against their formal specification [53]. Aktug and Naliuka introduce in

[38] Software-fault isolation.

2007 with *ConSpec* another language for the specification of policies based on security automata. In contrast to *Polymer*, it does not support the transformation capabilities of *Edit Automata*, since it reduces expressiveness in favor of a formal semantics. The already introduced SPoX system aims at the enforcement of aspect-oriented security policies on Java applications [100]. Dam et al. discuss the issue of multi-threaded Java-like programs for inline reference monitoring and introduce a class of *race-free* policies [57, 58]. Supporting such race-free policies is also of particular interest for future improvements to AppGuard.

Besides these Java oriented approaches, several systems were also developed for other platforms. Hamlen et al. introduced in 2006 the *Mobile* extension for the .NET Common Intermediate Language. It allows to specify security policies that are subsequently enforced via inline reference monitoring [102]. Vanoverberghe and Piessens propose a caller-side rewriting algorithm for the .NET bytecode MSIL [228]. Based on this algorithm, Vanoverberghe, Piessens, and colleagues have developed the S3MS.NET Run Time Monitor [61]. The tool supports both single-threaded and multi-threaded .NET programs and allows the specification of policies in several policy languages; however, its policies are in contrast to *Polymer* limited to the expressiveness of truncation automata. Shridhar and Hamlen propose to use model-checking for certifying the correctness of instrumented code, i.e., to prove that it fulfills the original policy [210]. Massacci and Siahaan propose optimizations based on automata modulo theory [139]. Given a security policy and a contract defining the trust boundary of a system, they refine the security to an optimized policy that assumes the contract as granted.

4.9.2 Security Extensions for Android

Besides the previously described approaches to inline reference monitoring that target at the deficiencies of the Android permission system, researchers have proposed several extensions to the Android operating system itself that also target at the coarse-grained and inflexible permission system. *Kirin* was introduced by Enck et al. in 2009 and aims at detecting and preventing the installation of third-party applications with dangerous permission combinations based on a pre-defined rule-set [64]. *Saint* by Ongtang et al. extends the Android permission system with new enforcement hooks to cover inter-application communication and to improve the protection of application interfaces [160]. The *Apex* system by Nauman et al. extends the Android permission system and allows users to revoke permissions of single applications and supports the specification of user-defined runtime constraints [153]. Another system called *CRePE*, which was introduced by Conti et al. in 2010, extends the Android middleware with new enforcement hooks to enable the enforcement of context-related policies [56]. An example of the authors with a typical context related policy is when users want to restrict the Bluetooth interface to be discovered at home or at work, but nowhere else. The *Porscha* system by Ongtang et al. adds content proxies and a corresponding reference monitor to the Android middleware to enable the enforcement of digital rights management through the operating system [159]. *AppFence* by Hornyack et al. was introduced in 2011 and extends Android with two novel privacy controls [106]: (1) The authors introduce the possibility to return shadow data instead of the original data in order to keep the

original data secret. This is an approach that we also follow in AppGuard in order to retain functionality of applications in cases where applications cannot be executed without specific data (for example, return a fake address book entry if the content is not relevant for the functionality itself). (2) The second mechanism taints data on (user) input and blocks connections when tainted data should be send off the phone. In 2012, Schreckling et al. introduced *Constroid*, a new policy management framework for Android that enables the definition of fine-grained data-centric security policies [195]. The *Aquifer* [151] system as introduced by Nadkarni and Enck in 2013 allows developers to define fine-grained policies for typical user workflows in order to protect sensitive data across applications.

Another line of research introduces more generic security frameworks for the Android operating system that aim at improving the access control system or in general the Android security model to enforce complex and fine-grained security and privacy policies. As already described in the Android primer (cf. Section 4.4), Android enforces permissions requested at the middleware layer at the kernel layer based on the standard discretionary access control of the Linux kernel. Smalley and Craig introduce *Security Enhanced (SE) Android* [207] based on *SELinux* [199] to provide mandatory access control at the kernel-layer and through middleware extensions also at the middleware layer. *Flaskdroid* [34] by Bugiel et al. follows the lines of *SE Android* and introduces a generic security architecture to provide mandatory access control for both the kernel and the middleware layer. *Flaskdroid* allows to instantiate previous security solutions such as *Saint* [160] and bases itself on SE Android [207]. In follow-up work on *Flaskdroid*, Heuser et al. developed with *ASM* [104] a programmable interface for implementing security extensions in form of security applications. Backes et al. also followed up on *Flaskdroid* and developed the Android Security Framework (ASF) [S3, S4], which introduces a new security API to facilitate the development of novel security extensions that either complement the existing Android security framework or that even partially replace it. ASF demonstrates its efficiency by implementing previous work such as *Flaskdroid* [34], *CRePE* [56], *XManDroid* [36], and even *AppGuard* [P4] as security modules. Zhou et al. propose with the *TISSA* extension a privacy mode for smartphones in which users can enforce a fine-grained access control for third-party applications accessing personal information such as contacts or location data [246]. *Kynoid* was introduced by Schreckling et al. and allows users the real-time enforcement of fine-grained and data-centric security policies on Android [196]. The system builds up on *TaintDroid* by Enck et al. [65]. Fragkaki et al. develop a formal framework (at the granularity of Android components and content providers) for analyzing Android's security mechanisms [76]. In addition, they introduce the *SORBET* system, which is designed to improve on the current permission system [76]. It integrates a reference monitor on top of the Android `ActivityManager` to allow developers the specification of secrecy and integrity policies. The goal is to mitigate undesired flows and privilege-escalation attacks. AppGuard aims with its secrecy policies also at the prevention of undesired information flows by fully blocking access to sensitive data upfront a potential declassification, however, with the advantage of not requiring changes to the operating system. In 2013, Jia et al. introduced a new system for the runtime enforcement of information-flow properties on Android by extending Android's `ActivityManager` [117]. The system uses labels stored

in the applications manifest file to express information-flow policies.

Trustdroid by Bugiel at al. introduces a security framework for Android that enables domain isolation at the middleware layer, at the kernel layer, and at the network layer of Android systems [35]. *KNOX* by Samsung goes in a similar direction and constitutes a platform security extension for Android [191] that targets at enterprises and integrates a variety of improvements: it hardens the operating system by implementing Secure Boot, a TrustZone-based integrity monitoring, and integrating *SE Android* [207]. Moreover, it ships with a variety of extensions to facility a secure device management and features like per-application VPNs, dedicated application containers and the enforcement of data encryption. Google and Samsung recently teamed up to integrate *KNOX* partially into the upcoming Android L release [179].

4.9.3 Application Analysis and Malware Detection

In the following, we will briefly introduce several approaches to application analysis and malware detection on Android. These approaches can be of great value for systems like AppGuard, as they establish the knowledge-base for many meaningful security- and privacy-protecting policies. Avik Chaudhuri introduces a typed language and operational semantics to describe Android applications and to reason about information flow properties at the granularity of Android components [48]. This work forms the basis for *ScanDroid* [49], which is to the best of our knowledge, the first automated tool that targets at the analysis of information flows in Android applications. The system is based on *WALA* [220] and compares the specifications from the application manifest file with the actual behavior of the app. Enck et al. subsequently introduced the *TaintDroid* system [65]. It utilizes the idea of dynamic taint tracking to track information flows on Android systems in real-time with the goal to achieve a real-time privacy monitoring. The *Paranoid Android* system by Portokalidis et al. records application executions on Android devices and replays them in a cloud service with much less resource constraints to perform virus scanning and dynamic taint tracking. *Paranoid Android* has the focus on malware detection [174]. Bläsing et al. propose *AASandbox* [30], an approach capable of performing static and dynamic analysis of Android applications. Static analysis is performed on the bytecode of third-party applications, the dynamic analysis is subsequently performed in an extended Android version running in the standard Android Emulator from the SDK. The authors see *AASandbox* as a potential cloud service to improve the detection rate of Android anti-virus applications [30]. Gilbert et al. introduce the *AppInspector*, which aims at an automated security testing of third-party applications during the vetting process of today's app markets [85]. The system facilitates dynamic taint tracking to detect explicit information flows and tracks control dependencies to detect implicit flows. Batyuk et al. use static analysis techniques to detect malicious behavior of applications and summarize the findings in a human-readable report for the user [23]. Afterwards, the system patches potentially malicious behavior of the application through binary rewriting to achieve a benign application. A typical example provided by the authors is the replacement of unique identifiers by a random UUID. The presented system forms the basis for the *Androlyzer* Web

service[39]. Another approach with particular focus on Android malware detection is *Andromaly* [200] by Shabtai et al. The framework realizes a malware detection system that monitors features and events on the mobile device and applies machine learning anomaly detectors to classify the collected data as benign or malicious. According to the authors, the framework was only tested with artificial malware since other was not yet available. Zhou et al. introduced in 2012 the *Droidranger* system [245]. It includes two mechanisms to detect malware: the first performs behavioral foot-printing based on permissions to detect new samples of already known Android malware families. The second mechanism performs heuristics-based filtering to identify certain typical malware behaviors. The authors analyzed more than 200.000 applications from five app markets and found more than 200 malicious applications including 32 in the official Google market. *Crowdroid* by Burguera et al. [40] is a client-side application that monitors system calls to the Linux kernel and sends them to a remote server. The remote server performs behavior-based malware detection based on clustering traces retrieved from crowd-sourcing. *Bati* is a powerful static analysis framework by Erik Derr [T1] to analyze information flow properties of third-party applications. The system is under continuous development and its current version is thought of as a tool for providing the ground truth for application-specific AppGuard policies. In a similar direction goes the *FlowDroid* system. It is based on the *SUSI* tool, a guided machine-learning approach for detecting sources and sinks of sensitive information directly from the Android source code [180]. *FlowDroid* takes the information provided by *SUSI* and tries to detect, based on static analysis, information flows from sources to sinks [15]. Another very interesting approach called *CHABADA* was introduced in 2014 by Gorla et al. [92]. The system automatically clusters applications according to the descriptions in the market and analyzes for each cluster the usage of permission protected Android APIs. Based on an anomaly classification, the authors detect outliers regarding the API usage, which are at least applications with unexpected behavior and likely malware candidates.

4.9.4 Securing Inter-app Communication

Applications in Android have by default zero rights. Applications are executed in an individual sandbox as well as in their own virtual machine and their rights can be extended at install time by requesting permission. Communication between applications and application components is only possible via Android's `Binder`-based inter-process communication[40]. Although applications can protect their own APIs via extra permissions, Android's permission system does not really cover the inter-app communication, which causes severe problems such as confused-deputy and collusion attacks. Research has come up with several attacks and extensions to Android in order to secure the inter-app communication. Using the insights from this research, AppGuard is able to mitigate certain attacks until a system or application patch is available.

Davi et al. [59] show how it is possible to mount privilege escalation attacks against third-party applications on Android. The authors identify the root cause of these problems in deficiencies of Android's permission-based security model. The authors

[39] Available at: `https://www.androlyzer.com/`

[40] Except for side-channel communication via sockets, files, etc.

show that Android's security model cannot cope with transitive permission usage. Porter Felt et al. analyzed in 2011 permission re-delegation attacks and defenses [173]. A re-delegation of permissions occurs according to the authors' definition if a privileged application performs an action on behalf of an unprivileged application. The authors prove the feasibility of such attacks on Android and introduce a system called *IPC Inspection* to mitigate such attacks. Quire [63] addresses the problem of confused deputy attacks by introducing two new security mechanisms. The system tracks call-chains of inter process communication to provide provenance information to callees. Based on this information the callee can either drop its privileges to the caller privileges or continue with full privileges. The second mechanism introduces a lightweight signature scheme and allows applications to create signed messages that can easily be verified by all other apps on the smartphone. Grace et al. developed in 2012 a tool called *Woodpecker* [94], which facilitates an inter-procedural data flow analysis to detect applications that leak permissions via unprotected APIs. The authors analyzed the images of eight popular Android smartphones by HTC, Motorola, Samsung, as well as Google and found for 13 analyzed permissions eleven permissions that where leaked at least on one phone. *ComDroid* by Chin et al. targets at improving the analysis of inter application communication by analyzing bytecode [54]. The tool aims both at consumers and developers and should allow them to detect potential communication vulnerabilities in third-party applications. *XManDroid*[41] by Bugiel et al. [36] introduces a security framework to protect both against confused-deputy and collusion attacks. Octeau et al. introduce a novel and sound static analysis technique for the Android platform, which is called *Epicc* and targets at inter-component communication [158]. *Epicc* is based on Soot and relies, therefore, in contrast to *ComDroid* [54] on a transformation of Dalvik bytecode to Java bytecode. Klieber et al. introduce a new static taint analysis for Android. The authors combine and extend the analysis of *FlowDroid* [15] and *Epicc* to track inter-component and intra-component data flows in a set of applications [124].

[41] This work is based on the technical report by Bugiel et al., which introduces the name *XManDroid* [37].

5 Conclusion

In this dissertation we presented three tools that improve trust and privacy in already existing ecosystems. In all three scenarios, the tools improve the state-of-the-art trust and privacy guarantees that can be provided for already deployed systems.

The first tool is WebTrust and it provides users with the means to trust the progressive verification of the correct transmission and authorship of static, dynamic, and live-streamed content in HTTP-based communication. The system is fully backwards compatible and supports both proving the authorship in front of third parties (non-repudiation) as well as the novel concept of individual verifiability. WebTrust enables the active revocation of documents and integrates an efficient update mechanism for protected data. The system allows to efficiently serve content via Web caches and content distribution networks, which reduces the overall network congestion. We presented a prototype implementation of WebTrust and conducted a series of performance measurements to prove the feasibility of the approach.

Second, we presented X-pire! and X-pire 2.0, two implementations of a digital expiration date for images in social networks. Both systems achieve this goal by encrypting images and using a novel technique to robustly embed the encrypted images into valid JPEG-files. Background of the embedding is the goal of achieving compatibility with the existing infrastructure so that the container images with the embedded encryptions are able to survive the recompression of existing upload routines in social networks. X-pire! is a solution purely in software and poses trust assumptions on both the systems' users and the operating system. In particular, X-pire! assumes that legitimate viewers of protected images do not copy decrypted images or the corresponding keys. X-pire 2.0 overcomes this limitation and poses no assumptions on the trustworthiness of users by leveraging ARM's trusted computing framework named TrustZone to provide robust guarantees. X-pire 2.0 is a general publication framework for digital content and can easily be adapted for data types other than images, as well as for other publication platforms than social networks. Especially in cases where the publisher maintains the

publication platform, our approach is straightforward to adapt. We have implemented X-pire! as a Firefox browser extension and X-pire 2.0 for the Google Android platform and conducted performance measurements to demonstrate the efficiency of both approaches.

Finally we presented AppGuard, a powerful framework for the enforcement of security and privacy policies on unmodified Android devices. The system is based on inline reference monitoring and overcomes Android's deficiencies with the permission system: It enables the dynamic revocation of Android permissions. Furthermore, AppGuard allows to enforce complex, fine-grained, as well as stateful security and privacy policies on third-party applications. Another important feature is its capability to provide a policy-based mitigation of vulnerabilities both in third-party applications and the operating system. With AppGuard we introduce a novel policy language named EXSPoX, which constitutes an extended version of the SPoX language as introduced by Hamlen and Jones in 2008 [100]. EXSPoX describes security automata and provides the transformation capabilities of edit automata. Using EXSPoX, we can specify information flow control policies and ensure the confidentiality of data by specifying our secret separation policies. We have proven the great utility of AppGuard to protect the user's privacy by enforcing a set of predefined policies on a variety of real-world third-party applications for Android.

Bibliography

Author's Papers for this Thesis

[P1] J. Backes, M. Backes, M. Dürmuth, S. Gerling, and S. Lorenz. "X-pire! – A digital expiration date for images in social networks." In: *CoRR* abs/1112.2649 (2011).

[P2] M. Backes, S. Gerling, and P. von Styp-Rekowsky. "Gezielte Vergabe von App-Rechten in Android – Smartphones für den Arbeitsalltag sichern." In: *IT-Sicherheit Ausgabe* 1/2013 (2013).

[P3] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. "AppGuard – Enforcing User Requirements on Android Apps." In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 543–548.

[P4] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. "AppGuard – Fine-grained Policy Enforcement for Untrusted Android Applications." In: *Proceedings of the 8th International Workshop on Data Privacy Management (DPM 2013)*. Vol. 8247. Lecture Notes in Computer Science. Springer, 2014, pp. 213–231.

[P5] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. *AppGuard - Fine-grained Policy Enforcement for Untrusted Android Applications*. Tech. rep. A/02/2013. Saarland University, 2013.

[P6] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. *AppGuard - Real-time policy enforcement for third-party applications*. Tech. rep. A/02/2012. Saarland University, 2012.

[P7] M. Backes, R. Gerling, S. Gerling, S. Nürnberger, D. Schröder, and M. Simkin. "WebTrust – A Comprehensive Authenticity and Integrity Framework for HTTP." In: *Proceedings of the 12th International Conference on Applied Cryptography and Network Security (ACNS 2014)*. Vol. 8479. Lecture Notes in Computer Science. Springer, 2014, pp. 401–418.

[P8] M. Backes, S. Gerling, S. Lorenz, and S. Lukas. "X-pire 2.0 – A User-Controlled Expiration Date and Copy Protection Mechanism." In: *Proceedings of the 29th ACM Symposium on Applied Computing (SAC 2014)*. ACM, 2014.

[P9] S. Gerling and R. W. Gerling. "Wie realistisch ist ein "Recht auf Vergessenwerden"?" In: *Datenschutz und Datensicherheit* 37.7 (2013), pp. 445–446.

[P10] P. von Styp-Rekowsky, S. Gerling, M. Backes, and C. Hammer. "Callee-site Rewriting of Sealed System Libraries." In: *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS'13)*. Vol. 7781. Lecture Notes in Computer Science. Springer, 2013, pp. 33–41.

Other Papers of the Author

[S1] M. Backes, S. Bugiel, and S. Gerling. "Scippa: System-Centric IPC Provenance Provisioning on Android." In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[S2] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder. "Acoustic Side-Channel Attacks on Printers." In: *Proceedings of the 19th Usenix Security Symposium*. Usenix Association, 2010, pp. 307–322.

[S3] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. *Android Security Framework: Enabling Generic and Extensible Access Control on Android*. Tech. rep. A/01/2014. Saarland University, 2014.

[S4] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. "Android Security Framework: Extensible Multi-Layered Access Control on Android." In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[S5] S. Bugiel, E. Derr, S. Gerling, and C. Hammer. "Advances in Mobile Security." In: *Proceedings of the 8th Security Research Conference on Future Security*. Fraunhofer Verlag, 2013.

Supervised Theses

[T1] E. Derr. "Verifying the Internet Access of Android Applications." M.Sc. thesis. Saarland University, 2011.

[T2] G. Geßner. "AppGuard: Automatic Compilation of Security Specifications to Java Code." B.Sc. thesis. Saarland University, 2013.

[T3] S. Obser. "User-controlled Internet Connections in Android." M.Sc. thesis. Saarland University, 2011.

[T4] L. Teris. "Securing User-data in Android: A conceptual approach for consumer and enterprise usage." M.Sc. thesis. Saarland University, 2012.

Other References

[1] *ab - Apache HTTP server benchmarking tool - Apache HTTP Server*. Online at `http://httpd.apache.org/docs/current/programs/ab.html`. Accessed on: 18.05.2014.

[2] L. von Ahn, M. Blum, and J. Langford. "Telling Humans and Computers Apart Automatically." In: *Communications of the ACM* 47.2 (2004), pp. 56–60.

[3] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. "CAPTCHA: Using Hard AI Problems for Security." In: *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Technique (EUROCRYPT 2003)*. Vol. 2656. Springer, 2003, pp. 294–311.

[4] T. Alves and D. Felton. "TrustZone: Integrated Hardware and Software Security - Enabling Trusted Computing in Embedded Systems." In: *ARM Information Quarterly*. Vol. 3. 4. 2004.

[5] *Android Developers*. Online at http://developer.android.com/. Accessed on: 18.07.2014. 2014.

[6] Android.com. *Android*. Online at http://www.android.com. Accessed on: 28.07.2014. 2014.

[7] Android.com. *Android Security Overview | Android Developers*. Online at http://source.android.com/devices/tech/security/index.html. Accessed on: 29.07.2014. 2014.

[8] Android.com. *Application Fundamentals | Android Developers*. Online at http://developer.android.com/guide/components/fundamentals.html. Accessed on: 28.07.2014. 2014.

[9] Android.com. *Porting Android to Devices | Android Developers*. Online at https://source.android.com/devices/index.html. Accessed on: 29.07.2014. 2014.

[10] Android.com. *The Android Source Code | Android Developers*. Online at http://source.android.com/source/index.html. Accessed on: 29.07.2014. 2014.

[11] *Apache Tomcat*. Online at http://tomcat.apache.org. Accessed on: 23.06.2014. 2014.

[12] Apple Press Info. *iPhone Premieres This Friday Night at Apple Retail Stores*. Online at: http://www.apple.com/pr/library/2007/06/28iPhone-Premieres-This-Friday-Night-at-Apple-Retail-Stores.html. Accessed on: 01.08.2014. 2007.

[13] ARM Limited. *ARM Security Technology – Building a Secure System Using TrustZone Technology*. Online at http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. Accessed on: 09.09.2013. 2009.

[14] ARM Limited. *TrustZone - ARM*. Online at http://www.arm.com/products/processors/technologies/trustzone.php. Accessed on: 09.09.2013. 2013.

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps." In: *Proceedings of the 35th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014)*. ACM, 2014.

[16] *ASM - Home Page*. Online at http://asm.ow2.org. Accessed on: 22.07.2014. 2014.

[17] G. Ateniese and B. de Medeiros. *A Provably Secure Nyberg-Rueppel Signature Variant with Applications*. Cryptology ePrint Archive, Report 2004/093, Available online at `http://eprint.iacr.org/2004/093`. 2004.

[18] G. Ateniese and B. de Medeiros. "On the Key Exposure Problem in Chameleon Hashes." In: *Proceedings of the 4th International Conference on Security in Communication Networks (SCN 2004)*. Vol. 3352. Lecture Notes in Computer Science. Springer, 2004, pp. 165–179.

[19] M. Backes, S. Gerling, and P. von Styp-Rekowsky. *A Local Cross-Site Scripting Attack against Android Phones*. Online at `http://www.infsec.cs.uni-saarland.de/projects/android-vuln/android_xss.pdf`. 2011.

[20] M. Backes, A. Kate, M. Maffei, and K. Pecina. "ObliviAd: Provably Secure and Practical Online Behavioral Advertising." In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy (Oakland 2012)*. IEEE Computer Society, 2012, pp. 257–271.

[21] D. Barrera, H. G. Kayacık, P. C. van Oorschot, and A. Somayaji. "A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android." In: *Proceedings of the 17th ACM Conference on Computer and Communication Security (CCS 2010)*. ACM, 2010, pp. 73–84.

[22] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. L. Traon. "In-Vivo Bytecode Instrumentation for Improving Privacy on Android Smartphones in Uncertain Environments." In: *CoRR* abs/1208.4536v2 (2013).

[23] L. Batyuk, M. Herpich, S. A. Çamtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak. "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications." In: *Proceedings of the 6th International Conference on Malicious and Unwanted Software (MALWARE 2011)*. IEEE Computer Society, 2011, pp. 66–72.

[24] L. Bauer, J. Ligatti, and D. Walker. "More Enforceable Security Policies." In: *Proceedings of the 2002 Workshop on Foundations of Computer Security*. 2002.

[25] L. Bauer, J. Ligatti, and D. Walker. "Composing security policies with polymer." In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*. ACM, 2005, pp. 305–314.

[26] R. J. Bayardo and J. S. Sorensen. "Merkle tree authentication of HTTP responses." In: *Proceedings of the 14th International Conference on World Wide Web (WWW 2005)*. ACM, 2005, pp. 1182–1183.

[27] F. Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *Proceedings of the Usenix Annual Technical Conference 2005 (ATC 2005), Freenix Track*. Usenix Association, 2005, pp. 41–46.

[28] M. Bellare and P. Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols." In: *Proceedings of the 1st ACM Conference on Computer and Communication Security (CCS 1993)*. ACM, 1993, pp. 62–73.

[29] T. Berners-Lee, R. T. Fielding, and H. Frystyk. *RFC 1945 - Hypertext Transfer Protocol – HTTP/1.0*. Online at `http://tools.ietf.org/html/rfc1945`. 1996.

[30] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Çamtepe, and S. Albayrak. "An Android Application Sandbox system for suspicious software detection." In: *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE 2010)*. IEEE Computer Society, 2010, pp. 55–62.

[31] D. Boneh and R. J. Lipton. "A revocable backup system." In: *Proceedings of the 6th Usenix Security Symposium*. Usenix Association, 1996, pp. 91–96.

[32] *bouncycastle.org: The Legion of the Bouncy Castle*. Online at `http://www.bouncycastle.org/`. Accessed on: 22.06.2014. 2013.

[33] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC." In: *Proceedings of the 15th ACM Conference on Computer and Communication Security (CCS 2008)*. ACM, 2008, pp. 27–38.

[34] S. Bugiel, S. Heuser, and A.-R. Sadeghi. "Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies." In: *Proceedings of the 22nd Usenix Security Symposium*. Usenix Association, 2013.

[35] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. "Practical and Lightweight Domain Isolation on Android." In: *Proceedings of the 1st ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2011)*. ACM, 2011.

[36] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. "Towards Taming Privilege-Escalation Attacks on Android." In: *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*. The Internet Society, 2012.

[37] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*. Tech. rep. TR-2011-04. Technische Universität Darmstadt - Cased, 2011.

[38] Bundesministerium des Innern. *Rede zum Thema "Grundlagen für eine gemeinsame Netzpolitik der Zukunft" durch Dr. Thomas de Maizière, Bundesminister des Innern*. Online at `http://www.bmi.bund.de/SharedDocs/Reden/DE/2010/06/bm_netzpolitik.html`. Accessed on: 27.08.2013. 2010.

[39] Bundesministerium für Ernährung, Landwirtschaft und Verbraucherschutz. *Umfrage zu Haltung und Ausmaß der Internetnutzung von Unternehmen zur Vorauswahl bei Personalentscheidungen*. Online at `http://www.bmelv.de/cae/servlet/contentblob/641322/publicationFile/36232/InternetnutzungVorauswahlPersonalentscheidungen.pdf`. Accessed on: 27.08.2013. 2009.

[40] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. "Crowdroid: behavior-based malware detection system for Android." In: *Proceedings of the 1st ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2011)*. ACM, 2011, pp. 15–26.

[41] S. Burnett, N. Feamster, and S. Vempala. “Chipping Away at Censorship Firewalls with User-Generated Content.” In: *Proceedings of the 19th Usenix Security Symposium*. Usenix Association, 2010.

[42] E. Bursztein and S. Bethard. “Decaptcha: Breaking 75% of eBay Audio CAPTCHAs.” In: *Proceedings of the 3rd Usenix Workshop on Offensive Technologies (WOOT 2009)*. Usenix Association, 2009.

[43] E. Bursztein, M. Martin, and J. C. Mitchell. “Text-based CAPTCHA Strengths and Weaknesses.” In: *Proceedings of the 18th ACM Conference on Computer and Communication Security (CCS 2011)*. ACM, 2011, pp. 125–138.

[44] E. Bursztein, A. Moscicki, C. Fabry, S. Bethard, J. C. Mitchell, and D. Jurafsky. “Easy Does It: More Usable CAPTCHAs.” In: *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems (CHI 2014)*. ACM, 2014, pp. 2637–2646.

[45] Carnegie Mellon University. *The Official CAPTCHA Site*. Online at `http://www.captcha.net`. Accessed on: 09.05.2014. 2010.

[46] C. Castelluccia, E. D. Cristofaro, A. Francillon, and M. A. Kâafar. “EphPub: Toward robust Ephemeral Publishing.” In: *Proceedings of the 19th Annual IEEE International Conference on Network Protocols (ICNP 2011)*. IEEE Computer Society, 2011, pp. 165–175.

[47] D. Catalano, D. Fiore, and R. Gennaro. “Certificateless onion routing.” In: *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS 2009)*. ACM, 2009, pp. 151–160.

[48] A. Chaudhuri. “Language-Based Security on Android.” In: *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2009)*. ACM, 2009, pp. 1–7.

[49] A. Chaudhuri, A. Fuchs, and J. Foster. *SCanDroid: Automated Security Certification of Android Applications*. Tech. rep. CS-TR-4991. Accessed on: 18.07.2014. University of Maryland, 2009.

[50] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. “Return-Oriented Programming without Returns.” In: *Proceedings of CCS 2010*. ACM, 2010, pp. 559–572.

[51] K. Chellapilla and P. Y. Simard. “Using Machine Learning to Break Visual Human Interaction Proofs (HIPs).” In: *Proceedings of the 18th Annual Conference on Neural Information Processing Systems (NIPS 2004)*. 2004.

[52] B. X. Chen and N. Bilton. *Et Tu, Google? Android Apps Can Also Secretly Copy Photos*. Online at `http://bits.blogs.nytimes.com/2012/03/01/android-photos/`. Accessed on: 10.07.2014. 2012.

[53] F. Chen and G. Roşu. “Java-MOP: A Monitoring Oriented Programming Environment for Java.” In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 546–550.

[54] E. Chin, A. Porter Felt, K. Greenwood, and D. Wagner. "Analyzing inter-application communication in Android." In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011)*. ACM, 2011, pp. 239–252.

[55] T. Choi and M. G. Gouda. "HTTPI: An HTTP with Integrity." In: *Proceedings of the 20th International Conference on Computer Communications and Networks (ICCCN 2011)*. IEEE Computer Society, 2011, pp. 1–6.

[56] M. Conti, V. T. N. Nguyen, and B. Crispo. "CRePE: Context-Related Policy Enforcement for Android." In: *Proceedings of the 13th Information Security Conference (ISC 2010)*. Vol. 6531. Lecture Notes in Computer Science. Springer, 2010, pp. 331–345.

[57] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. "Provably Correct Inline Monitoring for Multithreaded Java-like Programs." In: *Journal of Computer Security* 18.1 (2010), pp. 37–59.

[58] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. "Security Monitor Inlining and Certification for Multithreaded Java." In: *Mathematical Structures in Computer Science* (2011).

[59] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. "Privilege Escalation Attacks on Android." In: *Proceedings of the 13th Information Security Conference (ISC 2010)*. Vol. 6531. Lecture Notes in Computer Science. Springer, 2010, pp. 346–360.

[60] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. "I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications." In: *Proceedings of the Mobile Security Technologies Workshop 2012 (MoST 2012)*. IEEE Computer Society, 2012.

[61] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. "The S3MS.NET Run Time Monitor." In: *Electronic Notes in Theoretical Computer Scienc* 253.5 (2009), pp. 153–159.

[62] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. "Flexible Authentication Of XML documents." In: *Proceedings of the 8th ACM Conference on Computer and Communication Security (CCS 2001)*. ACM, 2001, pp. 136–145.

[63] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. "QUIRE: Lightweight Provenance for Smart Phone Operating Systems." In: *Proceedings of the 20th Usenix Security Symposium*. Usenix Association, 2011.

[64] W. Enck, M. Ongtang, and P. McDaniel. "On lightweight mobile phone application certification." In: *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS 2009)*. ACM, 2009, pp. 235–245.

[65] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." In: *Proceedings of the 9th Usenix Symposium on Operating Systems Design and Implementation (OSDI 2010)*. Usenix Association, 2010, pp. 393–407.

[66] Ú. Erlingsson. "The Inlined Reference Monitor Approach to Security Policy Enforcement." PhD thesis. Cornell University, 2004.

[67] Ú. Erlingsson and F. B. Schneider. "IRM Enforcement of Java Stack Inspection." In: *Proceedings of the 2000 IEEE Symposium on Security and Privacy (Oakland 2000)*. IEEE Computer Society, 2000, pp. 246–255.

[68] Úlfar Erlingsson and F. B. Schneider. "SASI enforcement of security policies: a retrospective." In: *Proceedings of the 1999 Workshop on New Security Paradigms (NSPW 1999)*. 1999, pp. 87–95.

[69] European Commission. *Proposal for a REGULATION OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL on the protection of individuals with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation)*. Online at `http://ec.europa.eu/justice/data-protection/document/review2012/com_2012_11_en.pdf`. Accessed on: 27.08.2013. 2012.

[70] D. Evans and A. Twyman. "Flexible Policy-Directed Code Safety." In: *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Oakland 1998)*. IEEE Computer Society, 1999, pp. 32–45.

[71] Facebook. Online at `http://www.facebook.com`. Accessed on: 15.05.2014.

[72] H. Federrath, K.-P. Fuchs, D. Herrmann, D. Maier, F. Scheuer, and K. Wagner. "Grenzen des "digitalen Radiergummis"." In: *Datenschutz und Datensicherheit* 6 (2011), pp. 403–407.

[73] R. Fielding and J. Reschke. *RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. Online at `http://tools.ietf.org/html/rfc7230`. 2014.

[74] Flickr. Online at `http://www.flickr.com`. Accessed on: 15.05.2014.

[75] A. Fox and E. A. Brewer. "Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation." In: *Proceedings of the 5th International Conference on World Wide Web (WWW 1996)*. Elsevier, 1996, pp. 1445–1456.

[76] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. "Modeling and Enhancing Android's Permission System." In: *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS 2012)*. 2012.

[77] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. *RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication*. Online at `http://tools.ietf.org/html/rfc2617`. 1999.

[78] Gartner. *Gartner Says Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013*. Online at `http://www.gartner.com/newsroom/id/2665715`. Accessed on: 29.07.2014. 2014.

[79] C. Gaspard, S. Goldberg, W. Itani, E. Bertino, and C. Nita-Rotaru. "Sine: Cache-friendly integrity for the web." In: *Proceedings of the 5th IEEE Workshop on Secure Network Protocols (NPSec 2009)*. IEEE Computer Society, 2009, pp. 7–12.

[80] GD Graphics Library. Online at `http://www.libgd.org`. Accessed on: 15.05.2014.

[81] R. Geambasu, T. Kohno, A. Krishnamurthy, A. Levy, H. M. Levy, P. Gardner, and V. Moscaritolo. *New Directions for Self-destructing Data*. Tech. rep. UW-CSE-11-08-01. University of Washington, 2011.

[82] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. "Vanish: Increasing Data Privacy with Self-Destructing Data." In: *Proceedings of the 18th Usenix Security Symposium*. Usenix Association, 2009.

[83] R. Gennaro and P. Rohatgi. "How to sign digital streams." In: *Proceedings of Advances in Cryptology (CRYPTO 1997)*. Vol. 1294. Lecture Notes in Computer Science. Springer, 1997, pp. 180–197.

[84] C. Gibler, J. Crussell, J. Erickson, and H. Chen. "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale." In: *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (Trust 2012)*. Vol. 7344. Lecture Notes in Computer Science. Springer, 2012, pp. 291–307.

[85] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. "Vision: Automated Security Validation of Mobile Apps at App Markets." In: *Proceedings of the 2nd International Workshop on Mobile Cloud Computing and Services (MCS 2011)*. ACM, 2011, pp. 21–26.

[86] J. Gionta, P. Ning, and X. Zhang. "iHTTP: Efficient Authentication of Non-confidential HTTP Traffic." In: *Proceedings of the 10th International Conference on Applied Cryptography and Network Security (ACNS 2012)*. Springer, 2012, pp. 381–399.

[87] O. Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs." In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC 1987)*. ACM, 1987, pp. 182–194.

[88] Google. *recaptcha*. Online at `https://www.google.com/recaptcha/intro/index.html`. Accessed on: 18.05.2014. 2014.

[89] Google. *Search removal request under data protection law in Europe*. Online at `https://support.google.com/legal/contact/lr_eudpa?product=websearch&hl=en`. Accessed on: 31.05.2014. 2014.

[90] *Google Play*. Online at `https://play.google.com/store`. Accessed on: 13.07.2014. 2014.

[91] R. Gootee. *Evil Tea Timer*. Online at `https://github.com/ralphleon/EvilTeaTimer`. 2012.

[92] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. "Checking App Behavior Against App Descriptions." In: *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 2014, pp. 1025–1035.

[93] *Governor Signs Steinberg Bill Protecting Minors' Privacy on the Internet*. Online at http://sd06.senate.ca.gov/news/2013-09-23-governor-signs-steinberg-bill-protecting-minors-privacy-internet. Accessed on: 03.10.2013. 2013.

[94] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. "Systematic Detection of Capability Leaks in Stock Android Smartphones." In: *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*. The Internet Society, 2012.

[95] J. Greene. *Intel® Trusted Execution Technology: White Paper*. Online at http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf. Accessed on: 28.05.2014. 2013.

[96] S. Guha, K. Tang, and P. Francis. "NOYB: Privacy in Online Social Networks." In: *Proceedings of the 1st ACM SIGCOMM Workshop on Online Social Networks (WOSN 2008)*. ACM, 2008, pp. 49–54.

[97] C. M. Gutierrez and J. M. Turner. *The Keyed-Hash Message Authentication Code (HMAC) (FIPS PUB 198-1)*. Online at http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf. 2008.

[98] 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: *Proceedings of the 17th Usenix Security Symposium*. Usenix Association, 2008, pp. 45–60.

[99] E. Hamilton. *JPEG File Interchange Format*. Online at http://www.jpeg.org/public/jfif.pdf. Accessed on: 04.01.2014. 1992.

[100] K. W. Hamlen and M. Jones. "Aspect-oriented in-lined reference monitors." In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2008)*. ACM, 2008, pp. 11–20.

[101] K. W. Hamlen, M. M. Jones, and M. Sridhar. *Chekov: Aspect-oriented Runtime Monitor Certification via Model-checking*. Tech. rep. UTDCS-16-11. University of Texas at Dallas, 2011.

[102] K. W. Hamlen, G. Morrisett, and F. B. Schneider. "Certified In-lined Reference Monitoring on .NET." In: *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2006)*. ACM, 2006, pp. 7–16.

[103] C. Hass. *JPEG Chroma Subsampling*. Online at http://www.impulseadventure.com/photo/chroma-subsampling.html. 2008.

[104] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. "ASM: A Programmable Interface for Extending Android Security." In: *Proceedings of the 23rd Usenix Security Symposium*. Usenix Association, 2014.

[105] S. Hohenberger and B. Waters. "Realizing Hash-and-Sign Signatures under Standard Assumptions." In: *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Technique (EUROCRYPT 2009)*. Vol. 5479. Lecture Notes in Computer Science. Springer, 2009, pp. 333–350.

[106] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. ""These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications." In: *Proceedings of the 18th ACM Conference on Computer and Communication Security (CCS 2011)*. ACM, 2011, pp. 639–652.

[107] *htop - an interactive process viewer for Linux*. Online at `http://htop.sourceforge.net/`. Accessed on: 19.05.2014.

[108] D. A. Huffman. "A Method for The Construction of Minimum Redundancy Codes." In: *Proceedings of the Institute of Radio Engineers (IRE)* 40.9 (1952), pp. 1098–1101.

[109] W. H. W. Hussin, R. Edwards, and P. Coulton. "E-Pass Using DRM in Symbian v8 OS and TrustZone - Securing Vital Data on Mobile Devices." In: *Proceedings of the 2006 International Conference on Mobile Business (ICMB 2006)*. IEEE Computer Society, 2006.

[110] *IBM Cryptocards*. Online at `http://www-03.ibm.com/security/cryptocards/`. Accessed on: 29.05.2014. 2012.

[111] *ImageMagick: Convert, Edit, Or Compose Bitmap Images*. Online at `http://www.imagemagick.org`. Accessed on: 15.05.2014.

[112] *InfoCuria – Case-law of the Court of Justice – Case C-131/12*. Online at `http://curia.europa.eu/juris/document/document.jsf?text=&docid=152065&pageIndex=0&doclang=EN&mode=req&dir=&occ=first&part=1`. Accessed on: 07.08.2014. 2014.

[113] Information Sciences Institute University of Southern California. *RFC 793 - Transmission Control Protocol*. Online at `http://tools.ietf.org/html/rfc793`. 1981.

[114] N. S. A. Information Systems Security Organization. *Controlled Access Protection Profile, Version 1.d*. Online at `http://www.niap-ccevs.org/pp/pp_os_ca_v1.d.pdf`. Accessed on: 25.05.2014. 1999.

[115] J. Jeon, K. K. Micinski, J. A. Vaughan, N. Reddy, Y. Zhu, J. S. Foster, and T. Millstein. *Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android*. Tech. rep. CS-TR-5006. University of Maryland, Dec. 2011.

[116] *Jetty - Servlet Engine and Http Server*. Online at `http://www.eclipse.org/jetty/`. Accessed on: 18.05.2014.

[117] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. "Run-Time Enforcement of Information-Flow Properties on Android - (Extended Abstract)." In: *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS 2013)*. Vol. 8134. Lecture Notes in Computer Science. Springer, 2013, pp. 775–792.

[118] *Jimple (Soot API)*. Online at http://www.sable.mcgill.ca/soot/doc/soot/jimple/Jimple.html. Accessed on: 22.07.2014. 2014.

[119] John Bryson (U.S. Department of Commerce) and Patrick Gallagher (NIST). *Secure Hash Standard (SHS) (FIPS PUB 180-4)*. Online at http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf. 2012.

[120] S. Josefsson. *RFC 4648 - The Base16, Base32, and Base64 Data Encodings*. Online at http://tools.ietf.org/html/rfc4648. 2006.

[121] Independent JPEG Group. *libJPEG*. Online at http://www.ijg.org. Accessed on: 15.05.2014.

[122] *JPEG Standard, ISO/IEC 10918-1 | ITU-T Recommendation T.81*. Online at http://www.w3.org/Graphics/JPEG/itu-t81.pdf. 1993.

[123] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007. ISBN: 1584885513.

[124] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. "Android taint flow analysis for app sets." In: *Proceedings of the ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP 2014)*. ACM, 2014.

[125] B. Könings, J. Nickels, and F. Schaub. *Catching AuthTokens in the Wild - The Insecurity of Google's ClientLogin Protocol*. Tech. rep. Accessed on: 10.07.2014. Ulm University, 2011.

[126] H. Krawczyk and T. Rabin. *Chameleon Hashing and Signatures*. Online at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.3574&rep=rep1&type=pdf. Accessed on: 01.06.2014. 1997.

[127] H. Krawczyk and T. Rabin. "Chameleon Signatures." In: *Proceedings of the 7th Annual Network and Distributed System Security Symposium (NDSS 2000)*. The Internet Society, 2000.

[128] C. Lesniewski-Laas and M. F. Kaashoek. "SSL Splitting: Securely Serving Data from Untrusted Caches." In: *Proceedings of the 12th Usenix Security Symposium*. Usenix Association, 2003, pp. 187–199.

[129] C. Lesniewski-Laas and M. F. Kaashoek. "SSL splitting: Securely serving data from untrusted caches." In: *Computer Networks* 48.5 (2005), pp. 763–779.

[130] S. Li, S. A. H. Shah, M. A. U. Khan, S. A. Khayam, A.-R. Sadeghi, and R. Schmitz. "Breaking e-Banking CAPTCHAs." In: *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC 2010)*. IEEE Computer Society, 2010, pp. 171–180.

[131] *Lift*. Online at http://liftweb.net/. Accessed on: 18.05.2014.

[132] J. Ligatti, L. Bauer, and D. Walker. "Edit Automata: Enforcement Mechanisms for Run-time Security Policies." In: *International Journal of Information Security* 4.1–2 (2005), pp. 2–16.

[133] C.-Y. Lin and S.-F. Chang. "Generating robust digital signature for image/video authentication." In: *Proceedings of the 1st Workshop on Multimedia and Security at ACM Multimedia 1998*. Vol. 98. ACM, 1998, pp. 94–108.

[134] H. Liu, S. Saroiu, A. Wolman, and H. Raj. "Software abstractions for trusted sensors." In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys 2012)*. ACM, 2012, pp. 365–378.

[135] M. M. Lucas and N. Borisov. "fnyob." In: *Proceedings of the 7th Annual ACM Workshop on Privacy in the Electronic Society (WPES 2008)*. ACM, 2008, pp. 1–8.

[136] W. Luo, Q. Xie, and U. Hengartner. "FaceCloak: An Architecture for User Privacy on Social Networking Sites." In: *Proceedings of the IEEE International Conference on Privacy, Security, Risk and Trust 2009 (PASSAT 2009)*. IEEE Computer Society, 2009, pp. 26–33.

[137] H. Maruyama, K. Tamura, and N. Uramoto. *RFC 2803 - Digest Values for DOM (DOMHASH)*. Online at `http://tools.ietf.org/html/rfc2803`. 2000.

[138] L. Masinter. *RFC 2397 - The "data" URL scheme*. Online at `ftp://www.ietf.org/rfc/rfc2397.txt`. Accessed on: 25.05.2014. 1998.

[139] F. Massacci and I. Siahaan. "Optimizing IRM with Automata Modulo Theory." In: *Proceedings of the 5th International Workshop on Security and Trust Management (STM 2009)*. 2009.

[140] *Massive Security Issues with Apps*. Online at `https://www.sit.fraunhofer.de/en/news-events/latest/press-releases/details/news-article/lll/`. Accessed on: 17.07.2014. 2014.

[141] Max Fischer, Washington Post. *Syrian hackers claim AP hack that tipped stock market by $136 billion. Is it terrorism?* Online at `http://www.washingtonpost.com/blogs/worldviews/wp/2013/04/23/syrian-hackers-claim-ap-hack-that-tipped-stock-market-by-136-billion-is-it-terrorism/`. Accessed on: 09.01.2014. 2013.

[142] V. Mayer-Schönberger. *Useful Void: The Art of Forgetting in the Age of Ubiquitous Computing*. KSG Working Paper No. RWP07-022. Available at SSRN: `http://ssrn.com/abstract=976541`. 2007.

[143] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. "Flicker: An Execution Infrastructure for TCB Minimization." In: *Proceedings of the ACM European Conference in Computer Systems (EuroSys 2008)*. ACM, 2008.

[144] R. C. Merkle. "A Certified Digital Signature." In: *Proceedings of Advances in Cryptology (CRYPTO 1989)*. Vol. 435. Lecture Notes in Computer Science. Springer, 1989, pp. 218–238.

[145] V. S. Miller. "Use of Elliptic Curves in Cryptography." In: *Proceedings of Advances in Cryptology (CRYPTO 1985)*. Vol. 218. Lecture Notes in Computer Science. Springer, 1985, pp. 417–426.

[146] D. Morrill. *Announcing the Android 1.0 SDK, release 1.* Online at: http://android-developers.blogspot.de/2008/09/announcing-android-10-sdk-release-1.html. Accessed on: 28.07.2014. 2008.

[147] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation - Methods and Techniques.* Online at http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf. 2001.

[148] T. Moyer, K. R. B. Butler, J. Schiffman, P. D. McDaniel, and T. Jaeger. "Scalable Web Content Attestation." In: *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009).* IEEE Computer Society, 2009, pp. 95–104.

[149] T. Moyer, K. R. B. Butler, J. Schiffman, P. McDaniel, and T. Jaeger. "Scalable Web Content Attestation." In: *IEEE Transactions on Computers* 61.5 (2012), pp. 686–699.

[150] *Mozilla Developer Center: XUL/Events Mutation DOM events.* Online at https://developer.mozilla.org/en/XUL/Events#Mutation_DOM_events. Accessed on: 19.05.2014.

[151] A. Nadkarni and W. Enck. "Preventing accidental data disclosure in modern operating systems." In: *Proceedings of the 20th ACM Conference on Computer and Communication Security (CCS 2013).* ACM, 2013, pp. 1029–1042.

[152] S. Nair, M. Dashti, B. Crispo, and A. Tanenbaum. "A Hybrid PKI-IBC Based Ephemerizer System." In: *Proceedings of the IFIP TC-11 22nd International Information Security Conference (SEC 2007).* Springer, 2007, pp. 241–252.

[153] M. Nauman, S. Khan, and X. Zhang. "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints." In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2010).* ACM, 2010, pp. 328–332.

[154] V. D. Nguyen, Y.-W. Chow, and W. Susilo. "Breaking an Animated CAPTCHA Scheme." In: *Proceedings of the 10th International Conference on Applied Cryptography and Network Security (ACNS 2012).* Vol. 7341. Lecture Notes in Computer Science. Springer, 2012, pp. 12–29.

[155] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197).* Online at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. Accessed on: 25.05.2014. 2001.

[156] NIST. *Recommendation for Key Management.* Special Publication 800-57 Part 1 Rev. 3, Online at http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf. Accessed on: 04.07.2014. 2012.

[157] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* DRAFT FIPS PUB 202, Online at http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf. Accessed on: 04.07.2014. 2014.

[158] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. "Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis." In: *Proceedings of the 22nd Usenix Security Symposium*. 2013, pp. 543–558.

[159] M. Ongtang, K. R. B. Butler, and P. D. McDaniel. "Porscha: policy oriented secure content handling in Android." In: *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC 2010)*. ACM, 2010, pp. 221–230.

[160] M. Ongtang, S. E. McLaughlin, W. Enck, and P. McDaniel. "Semantically Rich Application-Centric Security in Android." In: *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009)*. ACM, 2009, pp. 340–349.

[161] *OpenDHT: A Publicly Accessible DHT Service*. Online at `http://www.opendht.org`. Accessed on: 29.05.2014.

[162] *OpenSSL: The Open Source toolkit for SSL/TLS*. Online at `http://www.openssl.org/`. Accessed on: 24.06.2014. 2014.

[163] Oracle. *Java Cryptography Architecture – Oracle Providers Documentation*. Online at `http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html`. Accessed on: 22.06.2014. 2014.

[164] G. W. Paper. *The Trusted Execution Environment: Delivering Enhanced Security at a Lower Cost to the Mobile Market*. Online at `http://www.globalplatform.org/documents/GlobalPlatform_TEE_White_Paper_Feb2011.pdf`. 2011.

[165] *Path - Quality Social Networking*. Online at `https://path.com/`. Accessed on: 13.07.2014. 2014.

[166] W. B. Pennebaker and J. L. Mitchell. *JPEG still image data compression standard*. Chapman and Hall, 1993, pp. 97–110.

[167] R. Perlman. "File System Design with Assured Delete." In: *Proceedings of the 3rd IEEE International Security in Storage Workshop (SISW 2005)*. IEEE Computer Society, 2005, pp. 83–88.

[168] R. Perlman. *The Ephemerizer: making data disappear*. Tech. rep. SMLI TR-2005-140. Sun Microsystems, Inc., 2005.

[169] R. Perlman. "The Ephemerizer: Making Data Disappear." In: *Journal of Information System Security* 1.1 (2005), pp. 51–68.

[170] A. Perrig, R. Canetti, D. Tygar, and D. Song. "Efficient authentication and signing of multicast streams over lossy channels." In: *Proceedings of the 2000 IEEE Symposium on Security and Privacy (Oakland 2000)*. IEEE Computer Society, 2000, pp. 56–73.

[171] A. Porter Felt, K. Greenwood, and D. Wagner. "The Effectiveness of Application Permissions." In: *Proceedings of the 2nd Usenix Conference on Web Application Development (WebApps 2011)*. Usenix Association, 2011.

[172] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. "Android Permissions Demystified." In: *Proceedings of the 18th ACM Conference on Computer and Communication Security (CCS 2011)*. ACM, 2011, pp. 627–638.

[173] A. Porter Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. "Permission Re-Delegation: Attacks and Defenses." In: *Proceedings of the 20th Usenix Security Symposium*. Usenix Association, 2011.

[174] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. "Paranoid Android: Versatile Protection For Smartphones." In: *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC 2010)*. ACM, 2010, pp. 347–356.

[175] *PostgreSQL: The world's most advanced open source database*. Online at `http://www.postgresql.org/`. Accessed on: 15.05.2014.

[176] ProtonMail. *Security Details*. Online at `https://protonmail.ch/pages/security_details.php`. Accessed on: 30.05.2014. 2014.

[177] N. Provos. "Defending Against Statistical Steganalysis." In: *Proceedings of the 10th Usenix Security Symposium*. Usenix Association, pp. 323–336.

[178] M.-R. Ra, R. Govindan, and A. Ortega. "P3: Towards Privacy-Preserving Photo Sharing." In: *Proceedings of the 10th Usenix Symposium on Networked Systems Design and Implementation (NSDI 2013)*. Usenix Association, 2013, pp. 515–528.

[179] S. Rajagopalan. *KNOX Contribution to Android: Accelerating Android in the Workplace*. Online at: `http://android-developers.blogspot.fr/2014/07/knox-contribution-to-android.html`. Accessed on: 23.07.2014. 2014.

[180] S. Rasthofer, S. Arzt, and E. Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks." In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014)*. The Internet Society, 2014.

[181] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. "DROIDFORCE: Enforcing Complex, Data-Centric, System-Wide Policies in Android." In: *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES 2014)*. IEEE Computer Society, 2014.

[182] *Recommendation ITU-R BT.601-7 | Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios*. Online at `https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf`. 2011.

[183] I. S. Reed and G. Salomon. "Polynomial codes over certain finite fields." In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), pp. 300–304.

[184] S. Reimann and M. Dürmuth. "Timed revocation of user data: long expiration times from existing infrastructure." In: *Proceedings of the 11th Annual ACM Workshop on Privacy in the Electronic Society (WPES 2012)*. ACM, 2012, pp. 65–74.

[185] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. "Detecting In-Flight Page Changes with Web Tripwires." In: *Proceedings of the 5th Usenix Symposium on Networked Systems Design and Implementation (NSDI 2008)*. Usenix Association, 2008, pp. 31–44.

[186] E. Rescorla. *RFC 2818 - HTTP Over TLS*. Online at `http://tools.ietf.org/html/rfc2818`. 2000.

[187] E. Rescorla and A. Schiffman. *RFC 2660 - The Secure HyperText Transfer Protocol*. Online at `http://tools.ietf.org/html/rfc2660`. 1999.

[188] R. L. Rivest, A. Shamir, and L. M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[189] M. Rogers. *The Bearer of BadNews*. Online at `https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/`. Accessed on: 01.08.2014. 2013.

[190] *Safer Internet Day*. Online at `http://www.saferinternetday.org`. Accessed on: 08.08.2014. 2014.

[191] Samsung Electronics Co. Ltd. *White Paper: Meet evolving enterprise mobility challenges with Samsung KNOX*. Online at `http://www.samsung.com/global/business/business-images/resource/white-paper/2014/02/Samsung-KNOX_Whitepaper_web_Feb.27.2014-0.pdf`. Accessed on: 18.07.2014. 2014.

[192] N. Santos, H. Raj, S. Saroiu, and A. Wolman. "Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications." In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*. ACM, 2014, pp. 67–80.

[193] D. Sarno. *Twitter stores full iPhone contact list for 18 months, after scan*. Online at `http://articles.latimes.com/2012/feb/14/business/la-fi-tn-twitter-contacts-20120214`. Accessed on: 10.07.2014. 2012.

[194] F. B. Schneider. "Enforceable Security Policies." In: *ACM Transactions on Information and System Security* 3.1 (2000), pp. 30–50.

[195] D. Schreckling, J. Posegga, and D. Hausknecht. "Constroid: data-centric access control for android." In: *Proceedings of the 27th ACM Symposium on Applied Computing (SAC 2012)*. ACM, 2012, pp. 1478–1485.

[196] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff. "Kynoid: Real-Time Enforcement of Fine-Grained, User-Defined, and Data-Centric Security Policies for Android." In: *Proceedings of the 6th IFIP WG 11.2 International Workshop on Information Security Theory and Practice (WISTP 2012)*. Vol. 7322. Lecture Notes in Computer Science. Springer, 2012, pp. 208–223.

[197] D. Schreckling, S. Huber, F. Höhne, and J. Posegga. "URANOS: User-Guided Rewriting for Plugin-Enabled ANdroid ApplicatiOn Security." In: *Information Security Theory and Practice. Security of Mobile and Cyber-Physical Systems*. Vol. 78store86. Lecture Notes in Computer Science. Springer, 2013, pp. 50–65.

[198] D. Schröder and H. Schröder. "Verifiable Data Streaming." In: *Proceedings of the 19th ACM Conference on Computer and Communication Security (CCS 2012)*. ACM, 2012, pp. 953–964.

[199] SELinux. *Main Page - SELinux Wiki*. Online at `http://selinuxproject.org/page/Main_Page`. Accessed on: 22.07.2014. 2014.

[200] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. ""Andromaly": a behavioral malware detection framework for android devices." In: *Journal of Intelligent Information Systems* 38.1 (2012), pp. 161–190.

[201] H. Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)." In: *Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS 2007*. ACM, 2007, pp. 552–561.

[202] A. Shamir. "How to Share a Secret." In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[203] A. Shamir and Y. Tauman. "Improved Online/Offline Signature Schemes." In: *Proceedings of Advances in Cryptology (CRYPTO 2001)*. Vol. 2139. Lecture Notes in Computer Science. Springer, 2001, pp. 355–367.

[204] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. "Virtual Machine Showdown: Stack Versus Registers." In: *Proceedings of the 1st International Conference on Virtual Execution Environments (VEE 2005)*. ACM, 2005, pp. 153–163.

[205] *Siege Home*. Online at `http://www.joedog.org/siege-home/`. Accessed on: 04.07.2014. 2014.

[206] K. Singh, H. J. Wang, A. Moshchuk, C. Jackson, and W. Lee. "Practical End-to-End Web Content Integrity." In: *Proceedings of the 21st International Conference on World Wide Web (WWW 2012)*. ACM, 2012, pp. 659–668.

[207] S. Smalley and R. Craig. "Security Enhanced (SE) Android: Bringing Flexible MAC to Android." In: *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS 2013)*. The Internet Society, 2013.

[208] *Snapchat*. Online at `http://www.snapchat.com`. Accessed on: 29.05.2014. 2013.

[209] *Soot: a Java Optimization Framework*. Online at `http://www.sable.mcgill.ca/soot/`. Accessed on: 22.07.2014. 2014.

[210] M. Sridhar and K. W. Hamlen. "Model-Checking In-Lined Reference Monitors." In: *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2010)*. 2010, pp. 312–327.

[211] A. Srivastava and A. Amitabh. "ATOM: A System for Building Customized Program Analysis Tools." In: *Proceedings of the 15th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1994)*. ACM, 1994, pp. 196–205.

[212] A. Stavrou, J. Voas, T. Karygiannis, and S. Quirolgico. "Building Security into Off-the-Shelf Smartphones." In: *Computer* 45.2 (2012), pp. 82–84.

[213] D. Steinberg. *California Law – Senate Bill No. 568 Privacy: Internet: minors. CHAPTER 336, An act to add Chapter 22.1 (commencing with Section 22580) to Division 8 of the Business and Professions Code, relating to the Internet.* Online at http://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201320140SB568. Accessed on: 03.10.2013. 2013.

[214] D. R. Stinson. *Cryptography – Theory and Practice*. Third Edition. Chapman & Hall, 2006. ISBN: 978-1-58488-508-5.

[215] J. Tam, J. Simsa, S. Hyde, and L. von Ahn. "Breaking Audio CAPTCHAs." In: *Proceedings of the 22nd Annual Conference on Neural Information Processing Systems (NIPS 2008)*. 2008, pp. 1625–1632.

[216] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman. "FADE: Secure Overlay Cloud Storage with File Assured Deletion." In: *6th Iternational ICST Conference on Security and Privacy in Communication Networks (SecureComm 2010)*. Vol. 50. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, 2010, pp. 380–397.

[217] *The Chromium Projects*. Online at http://www.chromium.org/. Accessed on: 23.06.2014. 2014.

[218] *The Scala Programming Language*. Online at http://www.scala-lang.org. Accessed on: 18.05.2014.

[219] *The Trusted Boot project ("tboot")*. Online at http://tboot.sourceforge.net/. Accessed on: 28.05.2014.

[220] *T.J. Watson Libraries for Analysis (WALA)*. Online at http://wala.sourceforge.net/wiki/index.php/Main_Page. Accessed on: 23.07.2014. 2014.

[221] D. Travlos. *Forbes.com: ARM Holdings and Qualcomm: The Winners in Mobile*. Online at http://www.forbes.com/sites/darcytravlos/2013/02/28/arm-holdings-and-qualcomm-the-winners-in-mobile/. Accessed on: 10.08.2014. 2013.

[222] Trusted Computing Group. *TPM Main Specification Level 2 Version 1.2, Revision 116*. Online at http://www.trustedcomputinggroup.org/resources/tpm_main_specification. Accessed on: 28.05.2014. 2011.

[223] Trusted Computing Group. *Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.07*. Online at http://www.trustedcomputinggroup.org/resources/tpm_library_specification. Accessed on: 28.05.2014. 2014.

[224] *Trusted Computing Group - Media Room - FAQ*. Online at http://www.trustedcomputinggroup.org/media_room/faqs. Accessed on: 26.05.2014.

[225] *Trusted Platform Module Standard, ISO/IEC 11889-1 - 11889-4*. 2009.

[226] H.-W. Tseng and C.-C. Chang. "High Capacity Data Hiding in JPEG-Compressed Images." In: *Informatica* 15.1 (2004), pp. 127–142.

[227] *Twitter*. Online at http://www.twitter.com. Accessed on: 10.06.2014. 2014.

[228] D. Vanoverberghe and F. Piessens. "A Caller-Side Inline Reference Monitor for an Object-Oriented Intermediate Language." In: *Proceedings of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 2008)*. 2008, pp. 240–258.

[229] T. Vidas, N. Christin, and L. F. Cranor. "Curbing Android Permission Creep." In: *Proceedings of the Workshop on Web 2.0 Security and Privacy 2011 (W2SP 2011)*. IEEE Computer Society, 2011.

[230] N. Vratonjic, J. Freudiger, and J.-P. Hubaux. "Integrity of the Web Content: The Case of Online Advertising." In: *Proceedings of the Usenix Workshop on Collaborative Methods for Security and Privacy (CollSec 2010)*. Usenix Association, 2010.

[231] *Vuze Bittorrent Client - The Most Powerful Bittorrent Software on Earth*. Online at `http://www.vuze.com/`. Accessed on: 29.05.2014.

[232] *W3: Usage Statistics of Image File Formats for Websites, October 2013*. Online at `http://w3techs.com/technologies/overview/image_format/all`. Accessed on: 28.10.2013. 2013.

[233] wer kennt wen. Online at `http://www.wer-kennt-wen.de`. Accessed on: 15.05.2014.

[234] WhatsApp. *WhatsApp FAQ - Why does WhatsApp use my phone number and my address book?* Online at `http://www.whatsapp.com/faq/en/general/20971813`. Accessed on: 10.07.2014. 2014.

[235] R. Wilkins and B. Richardson. *UEFI Secure Boot in Modern Computer Security Solutions*. Online at `http://www.uefi.org/sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2013.pdf`. Accessed on: 28.05.2014. 2013.

[236] P. Williams and R. Sion. "Usable PIR." In: *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008)*. The Internet Society, 2008.

[237] J. Winter. "Trusted computing building blocks for embedded linux-based ARM trustzone platforms." In: *Proceedings of the 3rd ACM workshop on Scalable trusted computing (STC 2008)*. ACM, 2008, pp. 21–30.

[238] J. Winter, P. Wiegele, M. Pirker, and R. Tögl. "A Flexible Software Development and Emulation Framework for ARM TrustZone." In: *Trusted Systems*. Vol. 7222. Lecture Notes in Computer Science. Springer, 2012, pp. 1–15.

[239] C. Wisniewski. *Path and Hipster iPhone apps leak sensitive data without notification*. Online at `http://nakedsecurity.sophos.com/2012/02/08/apple-mobile-apps-path-and-hipster-and-leak-sensitive-data-without-notification/`. Accessed on: 10.07.2014. 2012.

[240] S. Wolchok, O. S. Hofmann, N. Heninger, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel. "Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs." In: *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS 2010)*. The Internet Society, 2010.

[241] B. Womack. *Google Gets Removal Requests After EU Privacy Ruling*. Online at `http://www.bloomberg.com/news/2014-05-30/google-creates-special-committee-for-eu-privacy-ruling.html`. Accessed on: 07.08.2014. 2014.

[242] R. Xu, H. Saïdi, and R. Anderson. "Aurasium – Practical Policy Enforcement for Android Applications." In: *Proceedings of the 21st Usenix Security Symposium*. Usenix Association, 2012.

[243] J. Yan and A. S. E. Ahmad. "A Low-cost Attack on a Microsoft CAPTCHA." In: *Proceedings of the 15th ACM Conference on Computer and Communication Security (CCS 2008)*. ACM, 2008, pp. 543–554.

[244] J. Yan and A. S. E. Ahmad. "Breaking Visual CAPTCHAs with Naive Pattern Recognition Algorithms." In: *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE Computer Society, 2007, pp. 279–291.

[245] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets." In: *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*. The Internet Society, 2012.

[246] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. "Taming Information-Stealing Smartphone Applications (on Android)." In: *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (Trust 2011)*. Vol. 6740. Lecture Notes in Computer Science. Springer, 2011, pp. 93–107.