TRUSTED CHANNELS AND ROOTS OF TRUST IN DISTRIBUTED EMBEDDED SYSTEMS

by

Dipl.-Ing. Steffen Schulz

DISSERTATION

submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy / Doktor-Ingenieur

as part of a joined doctoral supervision (cotutelle) between

MACQUARIE UNIVERSITY
SYDNEY ~ AUSTRALIA

Department of Computing,
Faculty of Science
Macquarie University, Australia

Advisor: Vijay Varadharajan
Submission: October 13, 2013

RUHR-UNIVERSITÄT BOCHUM

Faculty for Electrical Engineering and Information Technology,
Ruhr-University Bochum, Germany

Advisor: Ahmad-Reza Sadeghi
Disputation: July 7, 2015

Darmstadt, 2016
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zusammenfassung</td>
<td>xi</td>
</tr>
<tr>
<td>Abstract</td>
<td>xiii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xv</td>
</tr>
<tr>
<td>I  Problem and Background</td>
<td>1</td>
</tr>
<tr>
<td>1  Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Security and Trust in Complex Embedded Systems</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Thesis Scope and Summary of Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Efficient and Secure Trusted Networks</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Trust Establishment In Low-Cost Devices</td>
<td>6</td>
</tr>
<tr>
<td>2  Background: Trusted Computing and IPsec VPNs</td>
<td>9</td>
</tr>
<tr>
<td>2.1 TCG Trusted Boot and TPM</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 Trusted Platform Module (TPM)</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 Chain of Trust</td>
<td>11</td>
</tr>
<tr>
<td>2.1.3 Sealing and Binding</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Trusted Execution Environments</td>
<td>13</td>
</tr>
<tr>
<td>2.2.1 Dynamic Root of Trust for Measurement (DRTM)</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2 Mobile Trusted Execution</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Remote Attestation</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1 TCG Binary Attestation</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2 Trusted Channels</td>
<td>16</td>
</tr>
<tr>
<td>2.3.3 Alternative Attestation Methods</td>
<td>16</td>
</tr>
<tr>
<td>2.4 IPsec and IPsec-based VPNs</td>
<td>18</td>
</tr>
<tr>
<td>2.4.1 The Internet Key Exchange Protocol</td>
<td>20</td>
</tr>
</tbody>
</table>
II  Efficiency and Security of Trusted Networks  

3  Trusted Virtual Networks  

3.1  Secure VPNs in Turaya Secure OS  
3.1.1  The VPN Security Service  
3.1.2  Runtime Configuration Changes in Turaya  
3.1.3  Requirements for Attestation in Secure VPNs  
3.2  Extending IKEv2 for Remote Attestation  
3.2.1  Remote Attestation in the IKE SA  
3.2.2  The Attestation Data Payload  
3.2.3  Extracting the Attestation Key  
3.2.4  Attestation Service Interface  
3.3  Security Considerations  
3.4  Comparison with TCG Trusted Network Connect  
3.4.1  Integrating TNC Client Server Protocols  
3.5  Use-Case: Trusted Virtual Domains (TVDs)  
3.5.1  Architecture Overview  
3.5.2  The VDC Demonstrator  
3.5.3  Hypervisor Abstraction and Management  
3.5.4  Secure Virtual Networks  
3.6  Lessons Learned  
3.6.1  Multiple Logical Networks per TVD  
3.6.2  Code Re-Use in Secure OS Designs  
3.6.3  Secure Resource Labeling  
3.6.4  Information Leakage and Covert Channels  
3.7  Related Work  
3.8  Summary  

4  Analysis and Mitigation of VPN Covert Channels  

4.1  Problem Setting and Adversary Model  
4.1.1  System Model and Terminology  
4.1.2  Adversary Model  
4.2  Covert Channels in IPsec  
4.2.1  Storage Channels  

2.5  Conclusion  

II  Efficiency and Security of Trusted Networks  

3  Trusted Virtual Networks  

3.1  Secure VPNs in Turaya Secure OS  
3.1.1  The VPN Security Service  
3.1.2  Runtime Configuration Changes in Turaya  
3.1.3  Requirements for Attestation in Secure VPNs  
3.2  Extending IKEv2 for Remote Attestation  
3.2.1  Remote Attestation in the IKE SA  
3.2.2  The Attestation Data Payload  
3.2.3  Extracting the Attestation Key  
3.2.4  Attestation Service Interface  
3.3  Security Considerations  
3.4  Comparison with TCG Trusted Network Connect  
3.4.1  Integrating TNC Client Server Protocols  
3.5  Use-Case: Trusted Virtual Domains (TVDs)  
3.5.1  Architecture Overview  
3.5.2  The VDC Demonstrator  
3.5.3  Hypervisor Abstraction and Management  
3.5.4  Secure Virtual Networks  
3.6  Lessons Learned  
3.6.1  Multiple Logical Networks per TVD  
3.6.2  Code Re-Use in Secure OS Designs  
3.6.3  Secure Resource Labeling  
3.6.4  Information Leakage and Covert Channels  
3.7  Related Work  
3.8  Summary  

4  Analysis and Mitigation of VPN Covert Channels  

4.1  Problem Setting and Adversary Model  
4.1.1  System Model and Terminology  
4.1.2  Adversary Model  
4.2  Covert Channels in IPsec  
4.2.1  Storage Channels  

vi
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.4</td>
<td>Cooperative Attestation</td>
<td>134</td>
</tr>
<tr>
<td>7.5</td>
<td>Security Considerations</td>
<td>135</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Correctness and Unforgeability</td>
<td>135</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Prover Authentication</td>
<td>135</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Prover Isolation</td>
<td>135</td>
</tr>
<tr>
<td>7.6</td>
<td>Related Work</td>
<td>137</td>
</tr>
<tr>
<td>7.7</td>
<td>Summary</td>
<td>137</td>
</tr>
<tr>
<td>8</td>
<td>Transaction Security with SoftCards</td>
<td>139</td>
</tr>
<tr>
<td>8.1</td>
<td>Softer Smartcards with Trusted Execution</td>
<td>140</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Requirements and Adversary Model</td>
<td>140</td>
</tr>
<tr>
<td>8.1.2</td>
<td>System Architecture</td>
<td>141</td>
</tr>
<tr>
<td>8.2</td>
<td>Component Interaction and Protocols</td>
<td>142</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Deployment</td>
<td>142</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Migration and Backup</td>
<td>142</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Token Operation</td>
<td>143</td>
</tr>
<tr>
<td>8.2.4</td>
<td>PIN Caching</td>
<td>145</td>
</tr>
<tr>
<td>8.3</td>
<td>Implementation and Results</td>
<td>146</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Performance Evaluation</td>
<td>146</td>
</tr>
<tr>
<td>8.4</td>
<td>Security Considerations</td>
<td>147</td>
</tr>
<tr>
<td>8.5</td>
<td>Related Work</td>
<td>149</td>
</tr>
<tr>
<td>8.6</td>
<td>Summary</td>
<td>149</td>
</tr>
<tr>
<td>9</td>
<td>The TrustLite Security Architecture</td>
<td>151</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>151</td>
</tr>
<tr>
<td>9.2</td>
<td>Problem and Assumptions</td>
<td>152</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Terminology</td>
<td>152</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Adversary Model</td>
<td>153</td>
</tr>
<tr>
<td>9.2.3</td>
<td>System Requirements</td>
<td>153</td>
</tr>
<tr>
<td>9.3</td>
<td>The TrustLite Security Architecture</td>
<td>154</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Architecture Overview</td>
<td>154</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Memory Protection Unit (MPU)</td>
<td>155</td>
</tr>
<tr>
<td>9.3.3</td>
<td>Peripherals Access</td>
<td>157</td>
</tr>
<tr>
<td>9.3.4</td>
<td>TrustLite Exception Engine</td>
<td>158</td>
</tr>
<tr>
<td>9.3.5</td>
<td>Secure Initialization and Reset</td>
<td>160</td>
</tr>
<tr>
<td>9.3.6</td>
<td>Important Instantiations</td>
<td>161</td>
</tr>
<tr>
<td>9.4</td>
<td>Trustlet Invocation and Communication</td>
<td>162</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>9.4.1 Memory Layout and Entry Vectors</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>9.4.2 Inter-Process Communication</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>9.5 Evaluation</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>9.5.1 Prototype Implementation</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>9.5.2 Hardware Extensions Cost</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>9.5.3 Runtime Overhead of Memory Protection</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>9.5.4 Runtime Overhead of Exception Handling</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>9.6 Security Considerations</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>9.7 Related Work</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>9.8 Limitations</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>9.9 Summary</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>10 Conclusion and Future Work</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>List of Publications</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>Bibliography</td>
<td>181</td>
<td></td>
</tr>
</tbody>
</table>
ZUSAMMENFASSUNG


In diesem Zusammenhang kann die Fähigkeit, den Sicherheitszustand von Geräten automatisch über das Netzwerk zu verifizieren und Datenverfügbarkeit an bestimmte, vertraute Systemzustände zu binden, fundamental zur Sicherheit von aus vielen Geräten zusammengesetzten Infrastrukturen beitragen. Existierende Ansätze in diesem Bereich zielen jedoch vor allem auf leistungsstarke Plattformen wie PCs und Server ab, und sind nicht ohne Weiteres auf Geräte übertragbar, die zu sehr geringen Kosten produziert werden und oft starken Ressourcenbeschränkungen unterliegen.


Im zweiten Teil der Arbeit befassen wir uns zunächst mit einer Analyse von Software-basierter Attestierung, einem aktuellen Ansatz zur Erzeugung von Zustandsbeweisen der keinerlei vertrauenswürdiger Hardware bedarf. Dabei unternehmen wir einen ersten Schritt zur Formalisierung fundamentaler Anforderungen von Software-basierter Attes-

Zusammenfassend bieten die in dieser Arbeit vorgestellten und analysierten Konzepte neue Einsichten und Ansätze zur Konstruktion und Interaktion sicherer eingebetteter Systeme, und liefert so einen fundamentalen Beitrag zur Absicherung zukünftiger IT-Infrastrukturen.
Abstract

Embedded devices are increasingly used to automate critical infrastructures such as industrial plant supervision, home automation, and also medical implants and car electronics. But with their increasing inter-dependency, criticality and complexity, embedded systems also become viable and valuable targets of attackers.

In this context, the ability to automatically validate the security of remote systems and bind data to known-good system states can provide fundamental security assurance and facilitate the secure operation of distributed embedded systems in modern IT infrastructures. However, current approaches typically target the range of servers, PCs and rich mobile systems, and preclude low-cost and resource-constrained devices.

In this thesis, we investigate and present new solutions for trusted computing functionality in distributed embedded systems. We particularly focus on trust establishment and trusted channels in resource-constrained, low-cost embedded systems, with often limited computational capabilities and power supply.

We first investigate the extension of secure channels into trusted virtual networks. For this purpose, we extend the Internet Key Exchange (IKE) protocol to exchange remote attestation reports and build virtual datacenter prototype which accommodates multiple isolated trust domains. We then investigate the problem of covert channels in IPsec-based VPNs and present the design and implementation of an efficient covert channel-resilient IPsec stack. We also evaluate the feasibility of traffic obfuscation and normalization on mobile devices, and present a framework for targeted traffic flow manipulation.

As another major problem, we consider trust establishment in low-cost, resource-constrained devices. We analyze software-based attestation, a recent approach to low-cost attestation and trusted execution, and present a systematic and simplified construction of the software attestation checksum. Furthermore, we propose an extension of software attestation with PUFs, yielding a low-cost trust anchor that provides remote attestation and potentially also basic tamper evidence. We study the capabilities of current trusted execution environments and present a new security architecture establishing remote attestation and trusted execution even on low-end, resource-constrained devices.

Overall, we believe that the presented techniques comprise fundamental steps in improving and automating the security management of distributed embedded systems, thus improving their overall resilience to attacks.

xiii
Acknowledgements

This thesis comprises several years of research and development effort at Ruhr-University Bochum, Macquarie University in Sydney and Technical University Darmstadt. During this time, I have found a lot of support, encouragement, and also friendship among my various colleagues and collaborators.

I would like to thank my advisors Ahmad-Reza Sadeghi and Vijay Varadharajan for their continuous support, and for providing the opportunity to work with excellent colleagues in a challenging research environment.

I also like to thank my various co-authors, colleagues and assistants for sharing the fun (and burden) of paper writing with me, in particular Frederick Armkn echt, Ferdinand Brasser, Sven Bugiel, Lucas Davi, Thomas Fischer, Patrick Koeb erl, Thomas Schneider, Christian Wachsmann and Maria Zhdanova. It has been a pleasure to work with you and I hope we find more opportunities to collaborate in the future.

Finally, I like to thank my parents Christine and Klaus-Peter for their continued support at a time when computers were considered just strange toys, Hedi for teaching me multiplication tables with ice cream and grapes, Andreas and Benjamin for the competition, and Maria, for everything else.
Part I

Problem and Background
Chapter 1

Introduction

Embedded electronic devices increasingly permeate our society, re-defining and shaping our every-day life and interaction. But as a result of this process, they are also becoming increasingly critical to the security and privacy of their users and society as a whole. Usage profiles, credentials and sensitive documents accumulate in various devices around us, from sneakers and media players to smart TVs and smartphones [86, 323]. Printers, network routers and other inconspicuous equipment is repeatedly subject of exploitation and abuse, jeopardizing the security of large IT infrastructures and the data entrusted to them [76, 87]. In more critical applications such as electronic cars, medical implants and industrial plant control systems, the security of tiny, resource-impoverished embedded devices can even be critical for the safety of users and bystanders [265, 327, 323].

Due to the fast-paced product development and the often unpredictable results of interconnecting different services and usages online, appropriate privacy and security mechanisms are often only integrated after damage has been done. However, as the state-of-the-art in security and privacy technology advances, many mechanisms have also matured and consolidated into widely deployed standards. Responsible system designers have a range of well-examined techniques at hand to protect the authenticity, confidentiality and accountability of stored or transmitted data.

A major group of new security mechanisms that has received much attention in the last several years can be broadly described as “trusted computing functionalities”. In addition to cryptographic data protection, these mechanisms also consider the software state of involved platforms before exposing them to sensitive data, thus enabling the protection of data against compromised communication endpoints. We can distinguish three fundamental security mechanisms: Sealing protects the confidentiality of data such that it can only be decrypted on authorized platforms that are in a particular software state. Attestation describes the process of reporting the software state to another party for verification. It is typically combined with secure channels to produce communication channels that not only provide data confidentiality and endpoint authenticity/authorization, but also assure the integrity of the endpoint’s software state. Finally, Trusted Execution describes the verifyable execution of a particular program in an isolated environment, assuring that the program is not being manipulated by other software during its execution.

While several industry players performed a first major attempt at realizing these mechanisms within the Trusted Computing Group, their application in large heterogenous...
systems and especially the consistent deployment down to low-end, resource-constrained embedded systems, remain unsolved.

1.1 Security and Trust in Complex Embedded Systems

With the increasing demand for automation and convenience and improving power efficiency and performance of devices, embedded systems are becoming ever more complex, inter-connected and inter-dependent. Today, cars feature complex internal communication networks for infotainment as well as safety-critical sensor monitoring and vehicle control. Traffic lights are coordinated using wireless networks and even washing machines and shoes can communicate with smartphone Apps.

Abstractly, these complex embedded systems can be described as heterogeneous sensor networks. They can include a number of sensing and acting components of various computational capabilities and aim for mostly automated and self-sustaining interaction and regulation of certain processes. Often, a summarized, higher-level reporting and tuning of the self-regulation is performed by more capable devices and human operators.

However, with the increasing availability and complexity of complex embedded systems, they are also increasingly targeted by attackers, ranging from small scale enthusiasts and activists to criminal organizations and governments. This is aggravated by the fact that they often employ large amounts of legacy technology that was not originally intended for networked operation, such as insecure communication protocols and software interfaces that were not vetted against malicious inputs. Moreover, embedded systems are often deployed in scenarios with an increased risk of physical attacks, such as Pay-TV receivers or controllers regulating performance of a car engine.

The ability to validate the software state of remote systems and bind the availability of data to known software states can provide a major improvement in the security of these systems, as they heavily rely on the interaction of individual embedded components and the integrity and reliability of the exchanged data. Hence, the practical integration of trusted computing functionality and trusted channels into heterogeneous networks and in particular to the very low-end, low-cost embedded systems, are important problems of current research and the focus of this thesis. As such, the analysis of the proposed mechanisms and concepts is often limited to informal arguments and experimental validation, and a thorough formal analysis is mostly left to future work.

1.2 Thesis Scope and Summary of Contributions

Following this introduction, Chapter 2 provides some background to trusted computing systems and IPsec-based Virtual Private Networks (VPNs). The actual contributions of this thesis concern two major areas. The secure and efficient interconnection of devices in trusted networks is covered in Part II, while the establishment of trusted functionality in low-cost embedded devices is considered in Part III. Both parts comprise selected and extended/revised editions of peer-reviewed publications by the author. In particular, although the respective referenced publications often feature an alphabetical
1.2. THESIS SCOPE AND SUMMARY OF CONTRIBUTIONS

or alphabetical-by-institute listing of authors, any parts used in this thesis are that of the author, except where indicated differently in the following list of contributions.

1.2.1 Efficient and Secure Trusted Networks

Trusted networks are essential for the communication and abstraction of complex embedded systems as autonomous, isolated entities. However, previously proposed designs have typically focused on re-using existing protocols, resulting in complex nested protocols designed primarily for large-scale network management and automation.

In Part II, we thus consider the efficient and flexible establishment of trusted channels for resource-constrained devices. We evaluate the use of VPNs to create virtual networks with remote attestation. We construct a modified IPsec stack that is more suitable for the creation of logically isolated overlay networks and investigate the practicality of traffic analysis and traffic normalization for detecting and countering the detection of smartphone tethering.

Design and Implementation of Trusted Virtual Networks

Many secure channel designs employ Transport Layer Security (TLS) due to its maturity and seamless interoperability as a higher layer protocol. However, large-scale VPNs often require more flexible solutions that allow redundant connections, multi-cast, real-time protocols for Voice over IP (VoIP) etc.

Chapter 3 investigates the viability of IPsec as a basis for extensible trusted virtual networks. Following our prior design of a secure minimal-complexity IPsec-based VPN service [269, 15], we extend the IPsec Internet Key Exchange version 2 (IKEv2) to transport remote attestation messages. Our approach allows a more efficient exchange of attestation data than the Trusted Network Connect (TNC) client-server messages standardized by the Trusted Computing Group (TCG), while at the same time also supporting the update and evaluation of the reported software states while the channel is running. We then evaluate the concept of trusted virtual networks, using them as a basis for Trusted Virtual Domains (TVDs) in the virtual data center demonstration prototype of the Open Trusted Computing project [230]. However, one of the main conclusions from this work is that the abstraction of VPNs as logically isolated networks is often problematic: it assumes a resilience to information leakage and side-channel attacks that is usually outside the scope of secure channel protocols. Indeed, a number of side- and covert channel attacks have surfaced once the approach of elastic clouds for flexible server and application hosting found widespread use (e.g., [339, 250]).

Results of this work have been published in [13, 7], and were used as part of a demonstration prototype in the final review of the Open Trusted Computing (OpenTC) project [230]. The demonstrator was developed in collaboration with several partners, however, the author implemented the L4-based virtual network infrastructure presented in this thesis.

Analysis and Mitigation of Covert Channels in VPNs

A number of previous works investigate, analyze and propose defenses against covert channels and information leakage in security systems. However, preventing covert chan-
CHAPTER 1. INTRODUCTION

Channels between legitimate participants in shared computation, storage and communication resources is usually hard, and often incurs a strong performance penalty on the network. More recent works typically focus on the weaker information leakage model, aiming to prevent side-channel and traffic analysis attacks.

In Chapter 4, we re-consider the problem of covert channels for VPNs. We perform the first comprehensive analysis of covert channels in VPNs. Assuming dedicated VPN gateways, VPNs have a different trust model from secure channels, enabling more sophisticated covert channel elimination techniques. Following our analysis we design and implement covert channel mitigation techniques to prevent any information leakage from the VPN. Furthermore, we propose and implement the first runtime algorithm for adaptive traffic enforcement, allowing the dynamic reduction of protocol overhead while assuring a predefined maximum information leakage rate. We evaluate our scheme with regard to throughput and transaction delay and analyze different performance/security trade-offs. We demonstrate the performance and practicality of our leakage-resilient IPsec VPN in a randomized simulation with up to 80 simultaneous users.

Results of this research have been published in [14, 18]. Early drafts of those works were discussed with Amir Herzberg and Haya Shulman, and Amir proposed the mitigation mechanism for Packet Drops (PktDrop). The source code of the prototype, including benchmarking scripts and analysis tools developed previously in [10, 11] are available online: https://code.google.com/p/ipsec-tfc/.

Practical Traffic Analysis and Normalization

The covert-channel resilient IPsec VPN presented in Chapter 4 currently provides the most comprehensive and efficient option for leakage-resilient communication on the network layer. However, in many cases it is sufficient to consider a weaker adversary, allowing us to focus on more efficient solutions that may also be reasonably deployed on resource-constrained embedded systems.

In Chapter 5 we consider the problem of tethering detection and detection countermeasures in contemporary mobile phone networks. We analyze and categorize potential tethering detection techniques based on the approximate cost of performing them in a large customer base, and design a traffic normalization framework which circumvents all low to medium cost detection techniques. We implement an Android App, realizing the proposed traffic normalization in network as well as application layer protocols. Our evaluation shows that traffic normalization incurs much less network delay and protocol overhead, and as a result consumes only half the time and energy than transferring the same amount of data using a VPN.

This work has been published as [17]. Maria Zhdanova assisted in the literature review and assessment of detection mechanisms based on machine learning.

1.2.2 Trust Establishment In Low-Cost Devices

In Part III, we investigate solutions for trust establishment and trusted execution on low-end/low-cost embedded devices. These platforms are characterized by their very low targeted production cost and small form factors, resulting in strong constraints in terms
of energy consumption, available memory and computational performance. Their diverse emerging usages often demand rich communication options and flexible software stacks, raising questions on security and privacy assurance.

In this context, we systematize and analyze software-based attestation as an alternative paradigm for low-cost attestation and trusted execution. We propose a security protocol combining software attestation and Physically Unclonable Functions (PUFs), yielding a new class of low-cost Root of Trust. We implement secure software smartcards, enabling transaction security for many existing applications. Finally, we present a new platform architecture based on execution-aware memory protection that supports trusted execution even on highly resource-constrained, low-cost devices.

Analysis and Systematic Design of Software-based Attestation

Software attestation is a promising approach to establish trust in resource-constrained embedded devices without any additional hardware. Several implementations and variations of software attestation have been proposed in the literature, often employing non-cryptographic Pseudo-Random Number Generators (PRNGs) and ad-hoc hash functions for better performance [273, 271, 121, 192]. However, no formal security analysis has been performed on the overall protocol and the use specific requirements on its underlying cryptographic primitives.

In Chapter 6 we perform the first comprehensive analysis of software-based attestation, capturing a new class of time-limited, key-less challenge-response protocols. We identify and formalize assumptions on the system and cryptographic primitives and perform a first step towards formally proving software-based attestation. Following this analysis, we present the first systematic design of a software-based attestation algorithm and demonstrate their improved performance and simplicity compared to prior ad-hoc constructions. The work also identifies several open problems, such as further refinement of the cryptographic requirements and compressibility of the measured software state.

This work was done in close collaboration with Frederik Armknecht and Christian Wachsmann, who performed most of the formal modeling and proofs based on the author's background in the field. Results of this work have been published as [39, 3].

PUFs as Trust Anchors for Remote Attestation

One of the major problems of software-based attestation is the assumed out-of-band authentication channel between prover and verifier. Specifically, it is assumed that the prover platform cannot be exchanged by another device, or be modified to increase computational capabilities such as overclocking or extending its memory [275, 273, 74]. On the other hand, Physically Unclonable Functions (PUFs) are low-cost hardware extensions that exploit inherent tolerances in the hardware manufacturing process to create device identification and authentication mechanisms that are hard to clone.

Chapter 7 thus considers the use of Physically Unclonable Functions (PUFs) as lightweight trust anchors in software-based attestation. While previous trust anchors were designed using secure co-processors or trusted software [185, 160, 307], we show that a trust anchor for software-based attestation must only provide the functionality of a fast Pseudo-Random Function (PRF). Through careful integration with the software-based
CHAPTER 1. INTRODUCTION

attestation checksum, such a PRF can authenticate the computation steps performed on the prover. Moreover, by realizing the PRF using PUFs, the attestation mechanism yields improved resilience to physical key extraction and cloning attacks. We implemented the scheme by instantiating Arbiter-PUFs on a Field-Programmable Gate Array (FPGA).

Results of this work have been published in [16, 180]. A first hardware implementation on FPGA was realized in close collaboration with Ünal Koçabas [8].

Transaction Security with SoftCards

Multiple vendors have deployed mechanisms for trusted execution, allowing isolated and verifiable execution of code independently from Operating System (OS) and software stack [154, 24, 30]. These mechanisms can be used not only to protect secret keys but also a complete transaction, such as signing a document or logging into a website [210, 64, 108].

In Chapter 8, we propose and evaluate the use of the well-known Public Key Cryptography Standard (PKCS) #11 as an abstraction for Trusted Execution Environments (TEEs). We implement a simple software token inside the TEE to validate a PIN entry and perform signatures, using the TEE to assure that the PIN is physically entered at the platform and neither PIN nor signature keys are disclosed to the main operating system. Due to the well-supported PKCS#11 interface, the resulting prototype supports a variety of use-cases such as network authentication, email signatures or document signing, without modifying the corresponding userspace applications. Moreover, as the tokens exist as encrypted software bound to the TEE, they can be transferred via network for automated deployment and synchronization with other platforms of the same user.

The work has been published in [5]. Ferdinand Brasser, Thomas Fischer and Atanas Filyanov performed most of the implementation work for a prototype demonstration at Financial Crypto and the 2012 European Research@Intel technology showcase. The source code is available online: https://code.google.com/p/softcards/.

The TrustLite Security Architecture

A major result of Chapter 8 is that practical TEEs should support secure user input/output (I/O), as well as fast interaction with TEE applications. However, currently available solutions for low-cost devices suffer from significant performance overhead or incur unacceptable hardware cost.

Our final contribution in Chapter 9 thus presents a new hardware and software architecture for low-cost embedded systems. Leveraging a novel execution-aware memory protection scheme, we present low-cost methods for isolated and measured execution of tasks on a platform. We show how such trusted tasks can interact through well-defined interfaces and co-exist with an untrusted OS in a preemptive multitasking environment. TrustLite applications can maintain application state while inactive, enabling fast interaction with untrusted applications. Finally, trusted tasks can be provided exclusive hardware access, enabling the construction of secure device drivers and user I/O.

The hardware implementation was performed by Patrick Koebel, whose extensive hardware engineering background also guided many design decisions. The design and implementation results were published in [9], and an FPGA-based prototype and technical talk were also presented at the Intel European Research and Innovation Conference 2013.
Chapter 2

Background: Trusted Computing and IPsec VPNs

The term “Trusted Computing” has long been used to describe concepts and requirements for building secure computer systems. The Trusted Computer System Evaluation Criteria [314], or “Orange Book”, is probably the most renowned artifact of that era.

Today, trusted computing is mainly associated with the concrete set of standards and specifications of the Trusted Computing Group (TCG), a large industry consortium comprising several major hardware and software vendors. As part of this effort, several new security mechanisms and components have been added into mainstream computing platforms, in particular the well-known Trusted Platform Module (TPM) [311]. The effort is accompanied and extended by a large academic research community which is exploring new approaches, mechanisms and architectures for improving the security of current and future computing systems.

In the following sections, we provide an overview of the main concepts and developments in modern trusted computing. After introducing the basic concepts in Section 2.1, we consider more recent advances in trusted execution in Section 2.2. We provide a more extensive overview of attestation and trusted channels in Section 2.3 and introduce basic mechanisms of IPsec-based Virtual Private Networks (VPNs) in Section 2.4.
CHAPTER 2. BACKGROUND: TRUSTED COMPUTING AND IPSEC VPNS

2.1 TCG Trusted Boot and TPM

Trusted Boot and TPM are at the core of the TCG Trusted Computing Architecture [301]. The idea is that program code and configuration data is measured and measurements are securely recorded in the TPM before the program or data is loaded. This way, an indicator of the current machine state is introduced which can be used to authorize local data access or report the system state to other parties. In the following we discuss the fundamental concept and approach in more detail.

2.1.1 Trusted Platform Module (TPM)

The TPM is a secure co-processor that acts as the core security enforcement point in the runtime system. It provides a secure random number generator, non-volatile storage, and dedicated cryptographic functions like common RSA key operations and SHA-1 hash functions. The current version 1.2 of the TPM specification provides additional support for secure monotonic counters, an extended notion of different security domains (localities) and Direct Anonymous Attestation (DAA) [61].

The TPM protects a hierarchy of cryptographic keys. They are usually restricted to their specific usages and marked as either migratable or non-migratable. A migratable key can be transferred from one TPM to another (e.g., for backup purposes) whereas a non-migratable key must never leave the TPM that created it. The TPM introduces three major types of keys:

**Endorsement Key (EK):** The EK is a non-migratable key that represents the identity of the TPM. It is created during the production of the TPM and certified by the respective vendor to be a unique key residing in a specification-compliant TPM. It can thus be used to vouch for messages and actions performed by the TPM, enabling other parties to trust these actions.

**Attestation Identity Key (AIK):** As the EK uniquely identifies a TPM, it is also privacy-sensitive. Hence, AIKs can be generated with external clearing agents to create temporary pseudonyms for the EK (see Section 2.3).

**Storage Root Key (SRK)** The SRK is the root key-encryption-key of the larger key hierarchy protected by the TPM. This way, a variety of keys with different type and purpose can be protected without specialized hardware support.

Apart from cryptographic keys, one of the main facilities provided by the TPM are its Platform Configuration Registers (PCRs). The current TPM version 1.2 supports 24 such registers, each capable of storing a 160 bit SHA-1 hash sum. Access to the PCRs is restricted in such a way that they can act as a secure auditing service. In particular, a PCR can be modified only by extending it, i.e., by merging a new value \( v \) with the existing PCR value \( PCR_i \) as \( PCR_{i+1} \leftarrow SHA1(PCR_i || v) \). To facilitate the reconstruction and validation of the PCR values, the ordered sequence of values \( v \) that have been extended into the individual PCRs is recorded in the Stored Measurement Log (SML).

Finally, consider that the function of the TPM, a set of cryptographic operations on protected data and keys, can also be realized in a software component that is somehow...
isolated from the execution of the regular OS and user software. Indeed, the TCG Mobile Trusted Module (MTM) specifications describe how mobile devices can be equipped with a logical TPM that features a subset of the TPM commands and may be implemented in software [307, 302]. Note also that several improvements and generalizations of the TPM feature set are currently in work, including flexibility in the choice of cryptographic algorithms, in form of the TPM version 2.0 specification [308].

2.1.2 Chain of Trust

As the TPM is designed to be a passive component, acting only based on the (often authenticated) commands it receives, its function fundamentally requires the cooperation of the platform software. In particular, it is important to realize that the recording of the software state through the PCRs is also driven by the individual software components during bootstrapping and running of the system. In a compromised system, any measurement value \( v \) may be extended into the PCRs by the OS to imitate the loading of some expected software state.

Hence, the Chain of Trust is a fundamental notion when assessing the state of a system. In particular, the SML must be validated starting with the first loaded software component, and only if that measurement can be recognized to belong to a trusted software entity, further measurements in the measurement log can be iteratively validated and trusted to have correctly measured the respective next measurement.

As a result, the measurement chain must be started by a first, implicitly trusted software component outside the (passive) TPM. This component is denoted as the Core Root of Trust for Measurement (CRTM) and is typically a part of the platform firmware, e.g., the Basic Input/Output System (BIOS). The CRTM may measure additional BIOS components which are then handed control of the Central Processing Unit (CPU). Subsequently, the bootloader is measured and started and in turn measures the initial Operating System (OS) components before starting them. Another consequence is that the maintenance of the Chain of Trust is voluntary and rather brittle: The platform owner and/or user may at any point decide to load a program that does not continue the chain of measurements. Furthermore, it is important to perform and record all relevant measurements, i.e., to measure all data that has a fundamental impact on how software behaves, such as configuration files, libraries and scripts. The set of software modules that were started before some particular component are considered the security-critical basis of that component, its Trusted Computing Base (TCB).

Limitations

An important aspect of the Chain of Trust is that it only acts as an indicator of the system state. The Chain of Trust only records load-time measurements, while running programs may be influenced (and compromised) based on the inputs they receive at runtime. Hence, the trust that can be assured based on the Chain of Trust is only as good as the security and reliability of the individual measured software components, i.e., the absence of security bugs in each measured component over the life-time of the system. This is aggravated by the fact that there is no direct support for determining how a measured and launched
software may influence the existing system state, i.e., if it takes full CPU control like a bootloader or has limited execution privilege granted by a background process, such as sandboxed Java application or a virtual machine. In the TCG Chain of Trust, such effects must be extracted from the SML.

Finally, due to the rapid software development life cycles and the enormous complexity of today's platform states, it is a daunting task to accurately measure and identify all relevant data and code for subsequent validation. As a fundamental first step, Integrity Measurement Architecture (IMA) and Policy-reduced IMA (PRIMA) [262, 263, 159] provide a more sophisticated measurement architecture based on a trusted OS kernel. The trustworthiness of the OS kernel can be validated by the TPM Chain of Trust, and provide extended facilities such as a larger number of (software) PCRs, reducing the TCB of a system based on its isolation policy, and resetting PCRs once a securely isolated compartment of programs has finished execution.

Several works have proposed OS architectures that better cooperate with the notion of trusted software, TCB and Chain of Trust (e.g., [239, 122, 104, 281, 144, 208]). They focus on strong modularization and isolation of software components to facilitate the modularization and mineralization of any software's respective TCB. This is achieved by leveraging a minimal security kernel and a set of carefully programmed security services which the bulk of userspace legacy applications rely on. However, while these approaches reduce the complexity of the TCB and have been shown to be practical in several application scenarios, it remains to be seen if that reduced complexity can be maintained in production-level systems, and if it delivers the desired security improvements. Current evidence indicates that the cost of producing a trustworthy security kernel and other security services is rather large and may still require additional research [178, 256]. However, once developed, such a software system can be reused and should not require much maintenance.

2.1.3 Sealing and Binding

Assuming that the TPM PCRs reasonably reflect the system state, the TPM can then restrict data access by linking its availability to a certain set of PCR values. This way, an application can ensure that its data is only available in the expected (known, secure) system state.

In particular, when sealing data using the TPM, the software can specify a list of PCR indexes and values that will be encrypted together with the data by the TPM. When requesting a decryption of sealed data (unsealing), the TPM then checks not only if the corresponding key is available, but also if the specified PCR values match the currently set PCR values, and otherwise will not return the (internally already decrypted) data. In contrast, binding corresponds to the standard asymmetric encryption and is only limited by the availability of the corresponding encryption key.

Observe that the length of the data encrypted using RSA is fundamentally limited by the chosen RSA modulus and TPM hardware capabilities. Hence, the data is usually encrypted using a random symmetric key and only the key is then used in the binding or sealing operation. At the same time, the key used for the binding or sealing operation is often itself not permanently resident in the TPM but only loaded and decrypted
temporarily for a particular session, which itself also requires establishment and authorization. Since the TPM is also designed to be low-cost, interaction is often slow and requires several steps.

2.2 Trusted Execution Environments

A recent extension in many hardware platforms are Trusted Execution Environments (TEEs), a mechanism that executes code independently from previously executed software. Essentially, the CPU is re-initialized at runtime, so that a subsequently executed program (the payload) is not affected by previously executed, potentially malicious, software.

A major advantage of this technology is that the aforementioned chain of measurements can be reset, since previously executed software does not influence the security of the current software state anymore. For this purpose, the TPM was extended with a set of PCRs that can be reset by the CPU when entering the TEE. The CPU then measures the TEE payload and stores the measurements in the previously reset PCRs. This allows the use of shorter, more manageable measurement chains that are easier to verify (Dynamic Root of Trust for Measurement (DRTM)).

2.2.1 Dynamic Root of Trust for Measurement (DRTM)

The current desktop TEE designs are known as AMD Pacifica [24] and Intel Trusted Execution Technology (TXT) [153, 154]. They were designed based on the Dynamic Root of Trust for Measurement (DRTM) concept, i.e., to provide a soft platform reset that allows the start of a new (possibly compartmentalized in an own VM) Chain of Trust. The resulting construction is primarily suitable to launch a trusted OS without relying on previously launched code such as the BIOS or bootloader [168, 295, 208]. However, due to the strong desire for legacy-compatibility in the x86 architecture, this type of TEE still requires trust in the System Management Mode (SMM), which in turn is initialized by the BIOS and thus currently requires inspection of the original (non-dynamic) Chain of Trust [328].

Isolating Application Code with Flicker

Flicker [209, 210] is a software framework or OS driver for using the Intel and AMD TEEs. It is the first implementation that allows not only to start a new Chain of Trust out of a potentially compromised OS, but also to return to the previously running OS after execution of some security-critical code is finished.

As a consequence, Flicker is centered around the idea of securely executing a security-critical Piece of Application Logic (PAL) independently of a possibly compromised OS. For this purpose, an untrusted OS kernel module reserves memory for PAL execution, suspends OS execution and starts the TEE, providing the reserved memory as a parameter. Inside the TEE, a new Chain of Trust is started, measuring and loading a trusted Flicker component which then in turn measures and starts the PAL. As the OS execution
is suspended, the PAL has full control over all peripherals, allowing the establishment of secure user I/O that cannot be tampered with by a potentially compromised OS.

The scheme has been used to perform the signature or authentication operation of an application within a TEE [210], with the limitation however that the user is not provided with the required security indicators to distinguish a secure from an insecure session. A later work improves this situation by providing a uni-directional trusted path [108] for secure transaction confirmation. Using remote attestation, a remote system can then verify that the transaction was securely confirmed by a human user. Unfortunately, all of these approaches require substantial modification of individual applications or even the addition of a hypervisor.

2.2.2 Mobile Trusted Execution

Alternative forms of TEEs for mobile devices are realized by ARM TrustZone [30, 35] and Texas Instruments M-Shield [43]. In contrast to DRTM-style TEEs such as Intel TXT, these mobile TEE solutions provide a permanently running TEE that the CPU execution can switch to.

The root of trust is typically realized using Secure Boot, which initializes a trusted reference monitor, which controls TEE access and may also use the measure\&launch approach of Trusted Boot to dynamically load and validate further components.

The design allows switching between secure and non-secure execution more efficiently and flexibly and to maintain state in the TEE. In particular, TrustZone realizes the TEE simply as a "secure world" CPU mode that the software environment can be switched to. Additionally, TrustZone introduces a security bit on the address bus to signal if a particular transaction is coming from a secure or non-secure world component, thus enforcing a system-wide isolation between an arbitrary number of CPUs and peripherals.

This way, the TrustZone security framework allows an efficient parallel execution of secure and non-secure applications. Furthermore, peripherals may be extended to correctly handle secure and non-secure transactions in parallel, e.g., a display adapter may continue to update non-secure display content while showing a user dialog in foreground that was issued by some secure world application.

Texas Instruments M-Shield [43] provides similar hardware capabilities and in some variants directly builds up on TrustZone. However, while TrustZone is a fundamental framework for hardware security in System on Chips (SoCs), M-Shield follows the OMTP TR0 Trusted Environment recommendation [229] and focuses on compatibility and compliance of the manufactured devices. M-Shield has been used as a basis to implement a vendor-owned MTM [101]. To enable flexible use and extension of TEEs for secure user authentication, the authors of [99, 100] also realized On-board Credentials (ObC) on M-Shield, an open credential provisioning system.

Mobile TEEs are significantly more flexible and efficient than current DRTM-style TEEs, but currently tend to be locked from user access. Indeed, the approach of running all critical software in an isolated TEE mode is similar to virtualization with a trusted hypervisor, and may suffer from an increased TCB as the "trusted" hypervisor is extended with additional management features and device drivers for trusted peripherals.
2.3 Remote Attestation

Remote attestation is a protocol where a trusted computing platform *attests* its system state to another platform. We describe the basic concept of TCG binary attestation in Section 2.3.1, consider the various approaches to establish trusted channels in Section 2.3.2 and discuss some alternative approaches to attestation in Section 2.3.3.

2.3.1 TCG Binary Attestation

TCG remote attestation, or binary attestation, is the process where one platform (prover) securely transmits its TPM PCR values to another platform (verifier) for verification. More specifically, the verifier first issues a challenge to the prover, specifying the set of PCR values to be transmitted, as well as a nonce. The prover then requests the TPM to sign the requested PCR values and nonce with the secret key of some particular Attestation Identity Key (AIK). The signature is returned by the TPM, and transmitted together with the current Stored Measurement Log (SML), PCR values and the certified public key of the employed AIK to the verifier. The verifier validates the AIK certificate, the signature of the transmitted PCRs and then validates that the reported PCR values match to the state changes recorded in the SML.

The TCG specifications include two alternative schemes for AIK creation, one employing a trusted "Privacy CA" to provide secure pseudonymous AIKs and another one employing an untrusted anonymous signing service using Direct Anonymous Attestation (DAA).

Privacy-CA (PCA)

According to the TPM specification, the purpose of the Privacy-CA is to pseudonymously sign AIKs for the prover based on presented and validated EKs, while at the same time providing assurance to the verifier that the generated AIKs are only issued to validated EKs [241]. Hence, the PCA acts as a clearing house trusted by both, prover and verifier to perform its service correctly and securely.

However, to detect and expose compromised EKs the PCA will typically also maintain a log of generated AIKs and their associated EKs, resulting in a privacy risk for the prover. Hence, DAA was introduced to provide improved privacy in attestation.

Direct Anonymous Attestation (DAA)

DAA is an approach to generate verifiable AIKs using an anonymous credential scheme [61, 72, 201]. In particular, it allows the prover to anonymously authenticate to a DAA Issuer, which then issues a signing key that can be used (only) by the corresponding prover’s TPM to generate and sign its own AIKs. As a result, the communication with third parties is reduced, the AIKs are computationally hard to link to the corresponding EK and the DAA Issuer does not need to be trusted by the prover as it cannot identify the EK. However, a drawback is that detection and revocation of compromised EKs is much more costly [61].
2.3.2 Trusted Channels

Even before publication of the first TPM specifications, it was described how to assure endpoint integrity in secure channels, e.g., using hardware security modules [162].

With the emergence of the TPM specifications, more flexible and low-cost solutions have been proposed where large parts of the security is based on software components and cryptographic protocols. The works in [289] and [132] discuss in detail how TCG remote attestation can be linked to secure channels and suggest to use the credentials of the secure channel as input to the attestation algorithm. Alternatively, an X.509 certificate extension defined by the TCG [300] may be used to cryptographically link the attestation report to the respective secure channel endpoint [300, 37].

Another approach is to leverage trusted operating system architectures (Section 2.1) or trusted execution environments (Section 2.2). Leveraging these infrastructures one can create authentication credentials that are only available in a particular environment (platform state) associated with these keys. Such keys can then be used to encrypt data that can only be decrypted on a particular remote platform in a particular state, and is thus suitable as an offline trusted channel and extended access control, such as Digital Rights Management (DRM) [41, 42].

TCG Trusted Network Connect (TNC)

Within the TCG, the TNC group developed public specifications for combining existing secure channel protocols with remote attestation in a secure and inter-operable way [306]. For this purpose TNC extends the Extensible Authentication Protocol (EAP) [21], an authentication framework supported by many modern security protocols [21, 303]. Alternatively, a separate Transport Layer Security (TLS) channel may be established to tunnel the attestation protocol messages [309].

Leveraging the transports, TNC specifies TNC Client Server (TNCCS) as a common transport protocol to negotiate and transport different types of protocols for the exchange of actual attestation data between prover and verifier [310].

Unfortunately, while providing superior interoperability and flexibility compared to alternative trusted channel proposals, both the EAP and TLS transports result in significant delays and overhead during connection establishment. Since EAP itself is usually positioned as the authentication protocol, TNCCS can only be used as a tunneled “inner method”, following some other EAP authentication and incurring significant protocol overhead and additional message round-trips. Similarly, creating an additional TLS channel delays the connection establishment by multiple roundtrips required to establish and communicate over that channel.

2.3.3 Alternative Attestation Methods

Many works investigate alternative approaches to measuring and establishing trust in remote systems. In the following we focus on two major approaches in this field which have created significant interest in the community and were in turn subject of several research publications. Property-based Attestation aims to attest to higher level software
properties instead of binary measurements, while Software Attestation follows completely new approach to provide attestation on low-cost devices without trusted hardware.

**Property-Based and Semantic Attestation**

Due to the enormous software complexity in most contemporary systems, it is a hard organizational and technical (software engineering) problem to reliably measure and distinguish a secure from an insecure machine state [138, 259].

The fundamental idea of Property-based Attestation (PBA) [259, 242, 188, 79, 183] is that system security should be determined directly based on high-level system properties and security policies instead of low-level binary hashes of the deployed program binaries and configurations. As an example, the attestation service could report the availability of certain hardware security features such as the ability of the CPU to disable execution of writable memory (NX bit), if the latest software patches have been applied or if the current firewall configuration matches a certain security policy. This way, the verifier can directly request and verify properties that he is interested in, instead of reconstructing enforced security mechanisms based on the launched software configurations and versions. A similar approach is pursued in Semantic Remote Attestation [138], where the authors propose to attest to high-level properties of the program code language instead of its platform-specific binary representation.

Property-based attestation is more scalable in theory, providing new mechanisms for managing and reasoning with trust between devices [217, 119]. However, similar to binary attestation there are currently no good solutions for determining and extracting relevant security properties on a large scale, i.e., automatically and comprehensively. As a result, most current works simply assume that a trusted third party performs a secure assessment and association of the relevant software with its security properties [218].

**Software-based Attestation**

Software attestation is a trust establishment mechanism that allows a verifier to request and validate the integrity of the software state of another system (prover).

The scheme was originally proposed for resource-constrained embedded systems that cannot afford dedicated security hardware like secure memory [287, 88] or trusted co-processors [235]. It does not assume any secret (uncompromised) keys or trusted algorithms on the prover, following a radically different approach than most conventional security mechanisms.

For this purpose, software attestation exploits the computational constraints of the underlying hardware, such that any attempt of the prover to lie about its state leads to a wrong attestation response or a significantly increased response time. In particular, software attestation employs an iterative, checksum code that optimally uses the computational capacity of the prover to compute a randomized sample of the prover’s state. As illustrated in Figure 2.1, this is typically done by using the challenge $c$ to initialize a PRNG at the prover, which then generates random memory addresses $a_i$ for which the memory content $b_i$ should be merged into the checksum state. The process is repeated $k$ times in a simple loop that is assumed to be time-optimal. The final checksum state is returned to the verifier as the attestation response $r$. The verifier records the time it
takes the prover to compute the response based on the issued challenge. If the response was sent within the expected time frame, and the response matches the expected value based on the expected state and issued challenge $c$, the prover is accepted as correct.

The first conceptual description of assured remote code execution based on time constraints was designed as secure remote execution for mobile agents [146, 147]. The idea was extended in [171] to let the executed code securely report the software state and assure that this state is actually running on hardware, i.e., was not simulated. However, the scheme turned out to be vulnerable to multiple attacks: redundancy in the code and simple hardware manipulations could be used to forge valid software checksums and the CPU model is not sufficiently unique to identify a particular prover [275, 278].

The first detailed design and implementation of a software attestation protocol was presented in [275]. They reduced the scope of [171] to provide only a “local” attestation with some implicitly identified prover, e.g., using visual inspection, and also assume no hardware manipulations at the prover.

More recent works proposed several variations and extensions to software-based attestation, ranging from implementations for different platforms to more fundamental changes like repeated challenge-response procedures [160, 192], or using memory constraints [121, 318], self-modifying or obfuscated algorithms to relax the code optimality assumption [277, 129, 273].

Despite these improvements, the requirement to perform an out-of-band identification of the prover and to assure that malicious colluding parties do not interfere with the transmission is a fundamental problem for many practical scenarios. Hence, multiple works also consider the combination of software attestation with hardware trust anchors such as TPMs [268, 185] and Subscriber Identity Modules (SIMs) [160]. However, one may argue that such extensions contradict the idea of software-based attestation, and it is not clear why a secure co-processor used this way cannot be used directly to monitor and report the system state.

2.4 IPsec and IPsec-based VPNs

IPsec is a security architecture and protocol suite for the Internet Protocol (IP), extending it with integrity protection, authentication, confidentiality and (partial) replay
2.4. IPSEC AND IPSEC-BASED VPNS

IPv4 packet before applying ESP:

| orig IP hdr | (options) | TCP | Data |

IPv4 packet after applying ESP in transport mode:

| orig IP hdr | ESP | ESP | (+options) | Hdr | TCP | Data | Trailer | ICV |

| <---- encrypted -----> | <------ authenticated -------> |

IPv4 packet after applying ESP in tunnel mode:

| new IP hdr | ESP | orig IP hdr | ESP | (+options) | Hdr | (any options) | TCP | Data | Trailer | ICV |

| <------------ encrypted ------------> | <---------------- authenticated ----------------> |

Figure 2.2: Applying ESP protection in transport and tunnel mode, taken from [173].

IPsec also provides basic means for Traffic Flow Confidentiality (TFC), by partially obfuscating the patterns and endpoints of tunneled traffic in VPNs [173, 174].

The Internet Engineering Task Force (IETF) has published several specifications detailing IPsec operation in various scenarios. The overall IPsec architecture [174] specifies the interaction and security relations between components. The Internet Key Exchange (IKE) [170] is deployed as the core protocol for negotiation of keys, ciphersuites and protocols encapsulations between hosts, while actual data protection is provided by the Authenticated Header (AH) [172] and Encapsulated Security Payload (ESP) [173] protocols. Note that ESP provides both, authentication and encryption, while AH only provides authentication.

Two fundamental operation modes are supported in IPsec, transport and tunneling. In transport mode, only the payload of the IP packet is encapsulated in the ESP or AH protocol [172, 173], so that the outer IP header receives little or no protection, e.g., from IP spoofing. In tunnel mode, a tunneling gateway applies the ESP or AH protocols on the complete IP packet and generates a new outer IP header based on the configured tunneling setup and existing tunneled IP header information. This improves protection against IP header manipulation at the cost of additional protocol and processing overhead.

Both operation modes are shown for the case of ESP encapsulation in Figure 2.2. Observe that ESP includes not only a header but also a “trailer”, which includes a variable padding length field and an optional Integrity Check Value (ICV). In contrast, AH only uses a header.

Slightly different security properties are achieved by applying AH over ESP, in tunnel mode or transport mode or by using AH as well as ESP in different order. For example, AH in transport mode can still protect part of the IP header, while ESP only protects...
the encapsulated payload. Also, when using ESP in tunnel mode, an additional AH encapsulation may be used in transport mode on the inner IP packet to also provide authentication between hosts (Figure 2.3). However, apart from such side cases, transport mode is generally considered a subset of tunnel mode, and AH a subset of ESP. IPsec has received much criticism for its needless complexity in the past [107, 280, 89], which was partially addressed in later revisions of the standard by making AH optional and substantially simplifying IKE in IKEv2.

Several VPN solutions also deploy TLS [91] as a secure channel protocol instead of IPsec. The more recent Datagram TLS (DTLS) [249] also allows TLS over the simpler User Datagram Protocol (UDP), reducing the protocol overhead and undesired interference between inner and outer retransmission and rate adaption algorithms of the Transmission Control Protocol (TCP). However, while TLS-based VPNs are often easier to configure and provide a better usability, IPsec is significantly more flexible due to its lower layer operation, dedicated key exchange protocol and various extensions for mobility, multi-homing and multi-casting operation, among others.

We refer to [93] for a more detailed introduction and [236] for an overview of the cryptographic aspects of IPsec. The IKE protocol design and possible alternatives are discussed in [26], and an overview of the various standards and extensions of IPsec is available in [112].

2.4.1 The Internet Key Exchange Protocol

In the following we briefly introduce the Internet Key Exchange version 2 (IKEv2) protocol specified in [170]. We focus on the general protocol flow and message format in order to provide an understanding of the flexibility and potential for extensions such as presented in Chapter 3.

The IKE protocol was designed as a general protocol for negotiation of Security Associations (SAs), i.e., of keys, algorithms and other attributes needed to establish a secure channel. Its most prominent application is the negotiation of SAs for the IPsec AH and ESP protection protocols. When starting an IKE session, the protocol first performs a Diffie-Hellman key exchange, negotiating the first SA pair\(^1\) for securing the IKE channel (IKE SAs). Within this secure control channel, additional SAs for the actual

\(^1\)Since SAs are unidirectional, they are typically created and managed in pairs.
communication channels can be negotiated, refreshed or revoked simply by exchanging the corresponding messages (Child SAs).

**Protocol Flow.** Figure 2.4 depicts the basic message flow of IKEv2 and the required payloads in each exchange phase. The protocol works in pairs of messages, so-called exchanges. The first message of each exchange is sent by the *Initiator* and answered by the *Responder*. The standard IKEv2 protocol flow iterates through multiple phases, each of which consists of at least one message exchange with certain allowed payloads. The first phase, INIT, is used to exchange Diffie-Hellman public keys \((K_i, K_r)\) and to negotiate attributes of the IKE SA pair \((SA_1)\). The resulting (unauthenticated) shared secret \(K_{ir}\) is used to generate a session key \(SK\) that protects subsequent exchanges under the IKE SA \((encap_{SK}())\). The AUTH exchange is started in the second phase to mutually authenticate the endpoints of the IKE SAs and to negotiate a first set of Child SAs \((SA_2)\) that can be used for actual data transfer. After the authentication phase succeeded, the peers may use the established IKE channel secured by the IKE SAs to transmit additional notifications or to negotiate additional Child SAs for secure communication channels (INFO phase).

![Message Flow Diagram](image)

**Initiator**

\[
\begin{align*}
& N_i, K_i, SA_1 \\
& \leftarrow N_r, K_r, SA_1
\end{align*}
\]

**Responder**

\[
\begin{align*}
& SK = PRF(N_i, N_r, K_{ir}) \\
& \text{validate}(A_r) \\
& \leftarrow encap_{SK}(A_r, SA_2) \\
& \text{validate}(A_i)
\end{align*}
\]

\[
encap_{SK}(\cdots)
\]

Figure 2.4: Standard IKEv2 protocol flow with the IKEv2 Payloads for Diffie-Hellman key exchange \((K)\), SA proposals for IKE SA \((SA_1)\) and first Child SA \((SA_2)\), nonces \((N)\) and authentication of Initiator \((A_i)\) and Responder \((A_r)\).

**Message Format.** An IKEv2 message consists of the IKEv2 header followed by a dynamic list of payloads, each of which may again contain several substructures. Each IKEv2 payload starts with a *Generic Payload Header* that specifies the type and offset of the next payload in the message, as can be seen in Figure 2.5 (b). This design allows the Initiator and Responder to add optional or non-standard payloads to any message without interfering with the main handshake. If not supported or unexpected by the implementation of the receiver, payloads are simply ignored by jumping to the next available payload. However, the sender may also enforce processing of non-standard payloads by setting a “C” flag (critical) in that payload’s Generic Payload Header. In that case, the receiver must process the payload, or otherwise produce a corresponding error message.

As an example, we consider the *Security Association Payload* (SA Payload) in Figure 2.5 An SA Payload exchange is used to propose and select the algorithms and security
parameters for setting up a particular (unidirectional) SA. For this purpose, each SA Payload contains a list of SA Proposal Substructures that represent alternative choices for the SA to be negotiated. Each SA Proposal in turn contains a list of Transform Substructures that correspond to the available algorithms that can be negotiated as part of the SA. The Transform Substructures are categorized according to the available types of algorithms, e.g., algorithms for encryption, employed pseudo-random functions or authentication. Finally, each Transform Substructure can contain a list of Transform Attributes to signal the allowed parameters for the respective algorithm.

The recursive encoding of SA Proposals is illustrated in the example SA Payload in Figure 2.5 (a). It shows an SA Proposal structure that proposes the use of ESP with AES-CBC encryption and HMAC-SHA1-96 authentication. The AES-CBC algorithm is supplied with a Transform Attribute specifying possible key lengths, while the key length of HMAC-SHA1-96 is implicit in the algorithm (96 bit [187]). Order and numbering of structures is used to efficiently encode preferences and available combinations of algorithms. Also note that the type of the SA Proposal restricts its allowed Transform Substructures: While an SA Proposal for the AH protocol only contains the authentication Transform, an SA of type IKE SA contains at least four different types of Transform Substructures, negotiating attributes for encryption, authentication, Diffie-Hellman group and Pseudo-Random Function (PRF).

The design results in a highly extensible signaling protocol. New payload types, notifications and message exchanges can be added as easily as new ciphersuite algorithms and attributes.
2.5 Conclusion

The Trusted Computing initiative resulted in several new specifications and frameworks for delivering new security mechanisms to commodity computer systems.

But as the state-of-the-art in powerful server systems on the one hand and mobile/embedded platforms on the other hand advances, it becomes ever more difficult to scale the traditional Trusted Computing concepts and protocols in either direction. Especially for very low-end platforms which are often subject to strong constraints with regards to production cost and hardware resources, new concepts are required to deliver security beyond the old model of supervisor/usermode protection.

With remote attestation and trusted channels, these new security mechanisms can be extended into the network, and modern security protocols such as IPsec appear to be the perfect base components to deliver advanced security features across platforms and independently of the underlying communication media.
Part II

Efficiency and Security of Trusted Networks
Chapter 3

Trusted Virtual Networks

Secure communication between computer systems is typically established using technologies such as TLS [91] or IPsec [174]. Assuming that the communication endpoints are not compromised, these protocols ensure secure transmission of data and the authenticity of the communication endpoints. In many cases however, it is highly desirable to explicitly confirm and validate the trustworthiness of the involved remote endpoints, i.e., to use remote attestation to gain assurance that the remote system conform to a defined policy.

Unfortunately, a major issue with the TCG approach to attestation is the large number of possible states that modern computer systems can assume. Due to the complexity of today’s operating systems and applications, it is hard to create and maintain a list of all valid states of a system, or even to design trustworthy software in the first place. As a result, there is a large ongoing effort for minimizing the Trusted Computing Base (TCB) of a system, i.e., the number and the size of components that must be trusted. Projects like the Next Generation Secure Computing Base (NGSCB) [104], the European Multilaterally Secure Computing Base (EMSCB) [103], Open Trusted Computing (OpenTC) [230] and the various research prototypes such as sHype [261], Nizza [281] or Trustvisor [208] all attempt to realize a practical and scalable solution for trustworthy operating systems. By modularization and isolation of the system components, they aim to enhance the reliability and security of critical subsystems while maintaining compatibility and performance.

One particular solution recently presented in [15] is an IPsec-based VPN service that is optimized for security and low internal complexity. By using a microkernel-based operating system and by delegating all uncritical functionality like network card drivers and IP stack into isolated software modules, the so-called Secure VPN (sVPN) allows the creation of IPsec gateways with a small TCB.

In the following, we consider a flexible yet low-complexity extension of sVPN with remote attestation, and evaluate the resulting trusted VPN in a proof of concept prototype for Trusted Virtual Domains (TVDs). Although combinations of secure channels with attestation into trusted channels have been considered before [132, 42, 289, 124, 37, 303], no proposal exists that specifically targets IPsec VPNs, much less one that focuses on the fundamentally required low software complexity.

We start by introducing the Turaya Secure OS and its IPsec-based VPN security service in Section 3.1, and identify the requirements for extending IPsec VPNs with remote attestation. In Section 3.2 we propose an extension to the IPsec key exchange protocol,
CHAPTER 3. TRUSTED VIRTUAL NETWORKS

the Internet Key Exchange version 2 (IKEv2) [169]. For this purpose, we exploit that
IKEv2 regularly establishes its own secure control channel in addition to the actual IPsec
communication channels, enabling a continuous exchange of attestation data while the
IPsec connection is running. Following security considerations in Section 3.3, we compare
our solution with the existing Trusted Network Connect (TNC) standard in Section 3.4.
Finally, we present a proof of concept prototype employing trusted VPNS to implement
a virtual data center in Section 3.5. We present the network infrastructure deployed
in the prototype and consider the “lessons learned” of using trusted VPNS for virtual
infrastructures in Section 3.6.

3.1 Secure VPNS in Turaya Secure OS

The Turaya Secure OS is a prototype implementation of the PERSEUS security architec-
ture [258] for operating systems. The main idea is to externalize security-critical function-
ality into separate operating system services in the Trusted Services Layer (see Figure 3.1).
In the microkernel-based Turaya OS, these security services are normal processes from
perspective of the kernel. However, they are designed as self-contained isolated security
services with a minimal TCB. Other applications or even Virtual Machines (VMs) in the
system can leverage these services to perform their security-critical functionality. A typ-
ical example is the process of signing a digital document after it was created or received
through an application that is rather complex and often easily compromised, like a word
processor or web browser. In this case, a minimal application for viewing and signing of
documents can be used. It might additionally leverage a Trusted Storage service, which
will provide access to the users signing keys only after successful local attestation and a
Secure Graphical User Interface (GUI) service to provide trusted user I/O.

3.1.1 The VPN Security Service

We implement our sVPN service on top of the Turaya OS, since it was the only publicly
available modern mikrokernel at this time which also aimed to implement a secure TCB
for future Trusted Computing systems.

The two main design goals of sVPN are to minimize the complexity of the critical
subsystem in terms of components and code-size and to remain compatible with standard
IPsec in the major Virtual Private Network (VPN) use cases[269, 15]. As a result, sVPN
implements only a subset of IPsec, most notably a minimal IKEv2 key exchange and IPsec
packet processing using Encapsulated Security Payload (ESP) in tunnel mode. Further,
authentication based on X.509 certificates has been replaced with simple raw RSA key
authentication, which proved to be several magnitudes easier to implement and easier to
understand by users. Handling of uncritical functionality, like IP encapsulation, IP frag-
mentation and hardware drivers, is delegated to untrusted compartments. The security of
sVPN thus only depends on the security of the core IPsec functionality and some required
libraries for cryptographic primitives, threading, Inter-Process Communication (IPC) and
string operations. The sVPN service leverages the Trusted Storage to assure that the long-
term authentication keys are only disclosed to sVPN after local attestation. It optionally
leverages the trusted I/O of the Secure GUI service for authentication of local users.

Figure 3.1 shows a schematic overview of the integration of sVPN into the Turaya Secure OS. The sVPN service resides in the Trusted Services Layer together with other security services. In addition to the already mentioned Trusted Storage and Secure GUI, we see the Random Service which implements a secure Pseudo-Random Number Generator (PRNG).

### 3.1.2 Runtime Configuration Changes in Turaya

In this section, we consider changes in access control policy towards other system components in a modular operating system design. Goal is to further motivate efficient exchange of attestation data on remote channels and to illustrate how microkernel-based systems like Turaya enable scalable runtime attestation.

For attestation in the Turaya Secure OS, we distinguish between a static and dynamic TCB. The static TCB includes all components of the TCB that must be trusted by all applications and do not change at runtime. On the software side, this static TCB includes basic bootstrapping components, the Turaya microkernel and basic operating system services like memory management. The dynamic TCB in contrast comprises services that can be started and stopped as needed, but also of any other applications that the application in question can interact with. In this design, we can identify an individual dynamic TCB for each application in the system based on the set of components it is granted access to by the microkernel. The state and configuration of the dynamic TCB is tracked by the Property Manager (PropMgr) security service, which on demand generates authenticated attestation reports of the application’s individual dynamic TCB.

As an example, we refer to Figure 3.1 again where an Application VM in the Untrusted
Components layer is running on top of several security services in the Trusted Services layer. If access to the sVPN service is not allowed to this virtual machine, its dynamic TCB might contain just the Secure GUI service to allow secure switching between applications. Once access to sVPN is granted however, the dynamic TCB of this VM is extended to include the sVPN component and the dynamic TCB of the sVPN service. The PropMgr must recursively include all connected components until the static TCB is reached.

By observing changes to the access control table for IPC in the microkernel, the PropMgr can automatically derive the individual dynamic TCB of every task. Further, the PropMgr will be notified any interface with other applications can be considered for the remote attestation report.

This feature is particularly interesting if unanticipated changes to the local access policy are likely to occur during normal operation, e.g., access to hot-plug devices like mass storage or copy&paste to other applications. In such scenarios, we envision a service like sVPN to subscribe to PropMgr to receive notification of requests mediate them with remote peers connected via trusted channel. A simple practical use-case might be streaming video that is not supposed to be captured, or possibly confidential video conferences.

3.1.3 Requirements for Attestation in Secure VPNs

The security requirements for remote attestation protocols are not difficult to identify and many solutions are known [132, 42, 289, 124, 37, 303, 300]. However, we feel that minimal complexity and modularization is the best available approach to achieve scalable trustworthy systems. By isolating critical functionality from the remaining software, the TCB of a system is expected to become less complex and thus more reliable and also more stable over time.

Our goal is thus to integrate existing and future solutions for binary, property-based or even runtime attestation protocols with system designs that feature TCBs with high modularity and low complexity, like the Secure VPN (sVPN) design presented in [15]. For successful integration, we thus identify the following technical requirements for our protocol extension:

**R1 Security.** The attestation reports must be cryptographically linked to the endpoints of the associated secure channel to prevent a compromised endpoint from relaying attestation reports of other parties (cf. [132]).

**R2 Privacy.** Confidentiality of transferred attestation messages can be a requirement depending on the usage scenario, e.g., to comply with a company’s security policy.

**R3 Simplicity and Modularity.** As costs to validate and maintain software rise with its internal complexity, low software complexity is one of the main design goals of the sVPN architecture. To support this goal, the complexity added by our protocol extension should be minimal.

**R4 Communication Efficiency.** For general usability and to limit server load, our extension must support the exchange of attestation data with minimal additional protocol overhead and message roundtrips.
3.2. Extending IKEv2 for Remote Attestation

Our extension is implemented in three steps. First, we define an additional SA Transformation type Remote Attestation as an optional component of the IKE SA. This allows a peer to propose and select remote attestation as part of the negotiated set of algorithms. Secondly, we define a new IKE payload Attestation Data to tunnel the actual remote attestation data. Finally, we show how the actual attestation is securely linked to the IKE SA.

3.2.1 Remote Attestation in the IKE SA

As explained in Section 2.4.1, the IKEv2 protocol negotiates algorithms, key lengths and other attributes of an SA by formulating them in an ordered list of SA Proposal Substructures. For each SA negotiation, such a list is sent in an SA Payload by the Initiator. The Responder parses the SA Payload, selects a set of SA parameters and returns them in a corresponding response SA Payload. Figure 3.2 depict the format of the Transform and Transform Attribute Substructures and their use within the SA Proposals payloads.

Since the message format of IKEv2 is extensible by design and contains large ranges of identifiers that are “reserved for private use”, we can simply define a new Transform Substructure of type Remote Attestation and use its Transform ID field to identify up
CHAPTER 3. TRUSTED VIRTUAL NETWORKS

to $2^{16}$ specific remote attestation protocols. This makes the class of remote attestation algorithms available to the IKEv2 ciphersuite negotiation and, in case it is selected by both peers, allows us to define the additional semantics in Section 3.2.2 and 3.2.3.

Unfortunately, since such protocols can be quite complex and are still subject to research, they may exist in multiple variations. While the exact version can be negotiated within the attestation protocol, merging it with the SA negotiation step is more efficient and consistent. In particular it prevents the case where two parties agree on an attestation algorithm only to notice, multiple roundtrips later, that they do not support the same version of it.

We therefore also define a new Transform Attribute to encode protocol version numbers. Specifically, we use a simple Transform Attribute ($F = 1$ in Figure 3.2 (c)) and split the resulting 16 bit attribute value field into two 8 bit version numbers. The two numbers $V_{\text{min}}$ and $V_{\text{max}}$ are interpreted as an inclusive range of acceptable versions or, if $V_{\text{min}}$ is higher than $V_{\text{max}}$, as a negated version range. Similar to the Key Length Attribute, multiple Version Attributes can be included in a single Transform to encode intersections of version ranges. As an example, a Remote Attestation Transform with a Transform ID set to 1 might identify the property-based attestation protocol presented in [78] and an attached Version Attribute with $V_{\text{min}} = V_{\text{max}} = 2$ might identify the revised version of that protocol from [183].

This design allows an Initiator to propose an IKE SA with a remote attestation protocol in the same way it proposes different encryption or authentication algorithms. It can suggest multiple alternative protocols at once or make remote attestation optional by also including SA Proposals without a Remote Attestation Transform. The Responder has to select one complete set of parameters and express this set in its reply, or report an error that none of the proposals is acceptable. Selecting an appropriate Remote Attestation Transform thus imposes minimal overhead for the peers and is fully backwards compatible.

### 3.2.2 The Attestation Data Payload

Once a remote attestation protocol is negotiated, the messages of this protocol must be transmitted through IKEv2. To send these messages within the IKEv2 exchange, we have to define the layout and semantics of a payload structure that transports these messages. As creation and verification of attestation messages is a separate task that can be useful to many different applications besides IPsec, we let the actual attestation messages be handled by some separate Attestation Service which may or may not be part of the IKEv2 server.

As shown in Figure 3.3, the Attestation Data Payload (ADP) consists of the Generic Payload Header, a Data Length field and an opaque Attestation Data field. To rule out possible problems with duplicated or maliciously manipulated attestation requests as well as privacy concerns, ADPs must only be transmitted protected by the IKE SA, after the last IKE_Auth exchange succeeded. The opaque content of the payload may consist of multiple subsequent messages or logical channels, as for example supported by the TNC Client Server (TNCCS) protocol specified in [310]. The ADP defined here thus does not itself implement aggregation of multiple messages into a single payload but delegates this
3.2. EXTENDING IKEV2 FOR REMOTE ATTESTATION

Attestation Data Payload (ADP)

| Generic Payload | 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 |
| Header          | Next Payload | C | RESERVED | Payload Length |
| Header          |               |   | Data Length |
| Attestation Protocol | ~ <Attestation Data> ~ |
| Data            |               |   |

Figure 3.3: New payload to transport attestation messages through IKEv2.

functionality to the Attestation Service (AS). However, to also support simple attestation protocols efficiently the IKEv2 server may include multiple ADPs within a single IKEv2 message and thus transmit multiple queued attestation messages at once. In this case, the IKEv2 server is responsible for maintaining the order of Attestation Data messages. This order is already well-defined through the order of IKEv2 messages and the order of payloads within a message.

More sophisticated attestation mechanisms such as property-based attestation [78] may require the exchange of even larger attestation messages than the \(2^{16} - 1\) bytes (\(\approx 64\) KB) allowed maximum message size of a UDP datagram. Following the example of [303], we thus include the separate Data Length field to allow an overall attestation message of up to \(2^{32} - 1\) bytes or 4 GB to be fragmented over multiple ADPs. Since the order of IKEv2 messages and payloads within a message is well-defined and the secure channel provided by the IKE SA addresses packet loss and Denial of Service (DoS) attacks, reassembling such fragments is straightforward. Since IKEv2 messages are always exchanged in pairs, fragments are acknowledged with an empty IKEv2 message as defined in [169]. Defragmentation errors can be handled as IKEv2 payload parsing errors.

3.2.3 Extracting the Attestation Key

As specified in [169], the peers involved in the IKEv2 exchange initialize an internal Pseudo-Random Function (PRF) for each of the two negotiated IKE SAs. Based on the exchanged nonces and the shared Diffie-Hellman key \(K_{ir}\), the PRF is used to extract shared fresh symmetric keys for each algorithm of the two IKE SAs. The length of the keys depend on the respective negotiated algorithms and their attributes and is computed accordingly.

We extend this definition to create a shared Attestation Key \(AK\) if a remote attestation algorithm is selected as part of the IKE SA negotiation. As shown in Figure 3.4, we define the extraction process as \(SK\|AK := PRF(N_i, N_r, K_{ir})\). This is a simplified version of the extraction process defined in [169] which includes additional data into the PRF input and defines how to generate several keys for encryption, authentication etc. that we represent with \(SK\) here. Note that all the keys extracted in this manner are statistically independent from each other as long as the PRF is secure. Therefore, the order in which they are extracted is not relevant for their security. More importantly, this allows us
3.2.4 Attestation Service Interface

As flexibility is one of our main goals, we do not intend to restrict our protocol extension to one or more remote attestation protocols. Instead, we delegate interpretation, verification and creation of attestation data to an external generic Attestation Service (AS). In the following, we present the semantics of the IPC interface between the IKEv2 server and its (implicitly trusted) AS. A sample communication flow for a unilateral attestation of the Responder is illustrated in Figure 3.4.

**Connect()** After the last **AUTH** exchange succeeds, the IKEv2 server uses the **connect()** call to inform the AS that an attestation of the local platform or evaluation of a remote platform’s attestation report is needed. The call to the AS contains

1. the negotiated remote attestation protocol and attributes (**proto**),
2. the symmetric key **AK** of respective IKE SA,
3. the public key certificates, if used to authenticate the IKE SA and
4. an identifier of the corresponding network channel.

Figure 3.4: Modified IKEv2 protocol flow using new Transformation Structures in the \(SA1^*\) payloads (cf. Section 3.2.1) and additional \(ADP\) payloads that carry attestation messages \(m_1, m_2\) in the third protocol phase (cf. Section 3.2.2).
Note that in case of mutual attestation, the IKEv2 server will receive attestation requests and responses under the same IKE SA, and will thus also provide them to the AS under the same channel identifier.

**Data()** This call implements the exchange of attestation data between the local AS and IKEv2 server. It contains the channel identifier and the opaque attestation message $m$ that was received or is to be sent by the IKEv2 server.

**Grant()*/Deny()** The *deny()* call can be used by the Attestation Service (AS) at any time to revoke all Child and IKE SAs associated with the connection. The *grant()* call informs the IKEv2 server that the attestation succeeded and the associated Child SAs can be disclosed to the respective subsystems. The signaling of error messages or alternative attestation exchanges is the responsibility of the involved ASs.

**Close()** This call can be issued by both, AS and IKEv2 server to signal that the respective IPC connection and its associated IKE SA shall be closed. Any associated Child SAs are revoked (*revoke()*).

### 3.3 Security Considerations

In this section, we discuss the security of the trusted channel that can be established using the extension proposed above. Since our proposal is not restricted to a particular remote attestation protocol, we will use the unilateral challenge-response attestation shown in the third protocol phase of Figure 3.4 to show by example that our design achieves the following security goals.

**G1** Based on the security of the IKEv2 secure channel and careful choice of the attestation protocol, the IKEv2 extension allows the establishment of a trusted channel that meets our security requirements $R1$ and $R2$ of Section 3.1.3.

**G2** A compromise of the attested platform can be recognized in subsequent attestation exchanges if it is detected by the employed attestation protocol.

**Assumptions.** To show that $G1$ and $G2$ can be met with the protocol shown in Figure 3.4, we need the following additional assumptions.

**A1** *IKEv2 Security.* As specified in [169], the IKEv2 protocol establishes a secure channel based on the fresh shared keys stream that can be extracted from the PRF in the second phase. After the second phase succeeded, this channel provides an ordered exchange of authenticated and encrypted messages secure against packet loss, replay and downgrade or version rollback attacks.

**A2** *TCG PKI.* A Public Key Infrastructure (PKI) exists that allows the Initiator to validate the result of the attestation report of the Responder (e.g., result of $TPMQuote()$ operation plus measurement log) to gain assurance that the report is authentic and executed by a mutually trusted attester component.
**A3 Attestation.** The attestation mechanism (e.g., the TPMQuote() operation) on the Responder is secure in the sense that a *compromise* of the platform’s system state is reflected in subsequent attestation reports. In this context, *compromise* denotes any change to a platform’s state that violates the security policy of the Initiator.\(^1\)

**A4 IPsec Security** The security of the IKEv2 channel (A1) extends to the associated Child SAs and how they are used within IPsec. More precisely, the communication channels that are associated with a secure IKEv2 channel provide a secure channel with the properties negotiated in the Child SA negotiation, according to the security policies of Initiator and Responder (e.g., authentication, confidentiality, partial sequence integrity as specified in [173]).

**Adversary.** The adversary considered here is provided with two major attack vectors. Firstly, we assume that the network channel used by Initiator and Responder to communicate is fully under control of the attacker (V1). Secondly, we assume that the attacker can take control of the platform of the Responder at any time, even while the trusted channel is already established (V2), with the exception that the mutually trusted attester component of the platform remains integral.

**V1** The control of the network channel allows the attacker to launch downgrade, version rollback, replay, injection and many more attacks on IKEv2 and associated communication channels. Due to assumptions A1 and A4 however, these attacks are all prevented by the employed secure channel protocols. The adversary is still capable of launching a DoS attack, however, this is always possible with the given level of control on the network and thus trivial. In a more selected DoS attack, the adversary may attempt to either prevent the exchange of attestation messages after the secure channel is already established. However, as specified in Section 3.2.1, the use of attestation is negotiated in IKEv2 phase one and thus known and confirmed at both peers as soon as authentication succeeded. As specified in Section 3.2.4, the IKEv2 servers will thus wait for the decision of the Attestation Service to grant or deny the use of the actual associated communication channels. In face of selective DoS in later IKEv2 exchanges, the involved ASs can simply deny all further communication due to attestation timeout. As the attestation messages are only exchanged once the IKEv2 authentication phase succeeded, requirement R2 of goal G1 is met in case of V1.

**V2** When taking over control of a platform (in our example, that of the Responder) the adversary is free to modify or inspect its state, including, e.g., long term authentication keys and session keys for the secure channel established via IKEv2. However, by assumption A3 any such action is detected by the employed attestation protocol and reflected in subsequent attestation reports if it is relevant to the Initiator\(^2\).

---

\(^1\)Our goal is to show that the proposed system design is sound if a sufficiently secure attestation protocol is used, i.e., the attestation protocol is out of scope. We thus use this definition to obviate any discussion of the attestation scheme, including the problem of runtime attestation.

\(^2\)In practical systems, the platform configuration may be divided into isolated compartments that prevent the instant compromise of the whole system, thus allowing the relevant components to detect the compromise.
3.4 Comparison with TCG Trusted Network Connect

Based on the example attestation protocol illustrated in Figure 3.4, one can easily see how the Initiator can assure in this case that any subsequent attestation reports either report the compromise or fail the authentication:

**P1** When requesting an attestation report \((m_1 \text{ with requested property list } PCRlist)\), the freshness of the response is assured by including a fresh nonce \((N_1)\) in the request that must be used in combination with a one-way function when computing the response.

**P2** The attestation protocol must be designed such that attestation reports cannot be spoofed. In our example, this is achieved by combining the TCG TPMQuote operation with assumption \(A2\).

**P3** The remaining option for the adversary is to reflect the request of the Initiator to an uncompromised third party to receive a fresh and valid attestation report to answer the original request. This is prevented in our example by including the shared Attestation Key (AK) into the attestation report using a one-way function \(h()\). More precisely, the Responder hashes the nonce together with the shared Attestation Key of the associated connection and uses the result \(c = h(N_1, AK)\) as additional input to the digital signature computed in the TPMQuote() command. With growing bit length of \(AK\), the adversary has exponentially decreasing probability that the \(AK'\) used by the third party is the same as the \(AK\) used in the connection between Initiator and Responder, so that the attestation report is linked to the respective channel endpoint as required by R1. Alternatively, we also provide the certificate data used to authenticate the peers of the IKEv2 channel, thus providing additional ways to meet R1.

Finally, the adversary may choose not to send a response at all. However, the Initiator may simply signal the IKEv2 server to close the associated connections after some timeout to address this issue. Unfortunately, the violation of R2 is always possible (and thus trivial) if the peer that validates an attestation report is compromised and the attestation protocol discloses the state it attests to. This can be solved using privacy-preserving remote attestation protocols like [78, 183, 79].

With appropriate choice of the employed attestation protocol, our design thus achieves the security goals G1 and G2. The argument is easily extended to multilateral attestation and repeated attestation exchanges at runtime. In fact, any attestation protocol that follows the requirements in P1 to P3 meets our security goals under aforementioned assumptions A1 to A4 if only communicating using our extension.

### 3.4 Comparison with TCG Trusted Network Connect

The TCG TNC work group proposed an extensive infrastructure for collection, exchange and verification of attestation data [306]. Existing and future secure channel protocols are meant to be extended using either a dedicated TLS channel [309] or based on tunneled exchanges within the Extensible Authentication Protocol (EAP) framework [303]. While
the TNC framework is highly flexible and integrates well with EAP-based centralized network access control management, it fails to meet our requirements for simplicity and efficiency: The use of EAP imposes a significant protocol overhead in terms of roundtrips and message size and relies on the secure configuration and implementation of multiple additional protocol layers. For instance, IKEv2 can establish a secure control channel and generate session keys for content protection within two roundtrips. Any additional session key pairs can be generated in a single additional roundtrip. By comparison, the use of EAP involves negotiating and performing the tunneled EAP authentication method, such as TLS, and then negotiating and performing the EAP-TNC protocol as a sub-method of that authentication protocol [303]. The tunneled EAP-TNC protocol then in turn negotiates the attestation protocol to be used and performs the exchange of attestation data in the TNCCS protocol inside the EAP-TNC sub-method [310]. Overall, this approach encapsulates the required attestation data in at least 5 protocol layers and incurs at least 8 additional roundtrips to establish IPsec session keys, plus communication to central authentication services and possible packet fragmentation. Finally, since EAP is intended as an authentication framework which is only executed once when establishing a secure channel, it does not support the exchange of attestation data at runtime.

On the other hand, the IF-T Binding to TLS requires a dedicated TLS channel for the exchange of attestation messages. This allows exchanging additional attestation reports at runtime; however, the cost of implementing and negotiating Transport Layer Security (TLS) as well as the associated certificate management is considerable, and further complicated by the requirement to cryptographically link the attestation reports to the secure channel endpoints.

Overall, our solution to directly support attestation data as an extension of the IKEv2 protocol results in a much simpler and more flexible solution. It enables the extension of IKEv2 secure channels into trusted channels with as little as one additional roundtrip (depending on the employed attestation protocol), and does not require changes to existing EAP implementations.

3.4.1 Integrating TNC Client Server Protocols

As explained in Section 3.1.3, the primary goal of our proposal is the efficient and flexible transport attestation messages over the IKEv2 protocol. From perspective of the TNC architecture, our proposal can thus be seen as a new IF-T Binding to IKEv2 which leverages the existing secure channel.

In fact, our extension meets the requirements of the TNCCS protocol. Specifically, the requirements for Chunking, Transport and Security are met through transparent in-order transfer for messages of up to $2^{32} - 1$ bytes and the secure channel provided by the IKE SA. Our protocol extension can thus be used to transport TNCCS messages transparently, with one major caveat: As our design leverages the secure channel provided by IKEv2, exchanged attestation messages are only protected during transmission between the two involved IKEv2 servers. Our protocol does not explicitly support the case where (part of) the Attestation Service is on a remote system. However, where such a design is desired, the existing IPsec implementation can be used to configure additional secure tunnels towards the AS.
3.5 Use-Case: Trusted Virtual Domains (TVDs)

In the following we present the design and implementation of a TVD proof of concept prototype, describing the deployed components and mechanisms. Section 3.6 then summarizes the “lessons learned”, pointing out the major previously not considered problems when realizing trusted virtual infrastructures in practice.

A TVD [135, 67, 167, 52] is a coalition of virtual machines that trust each other based on a common security policy. The policy is uniformly enforced, independent of physical boundaries. TVDs build upon virtualization techniques to provide containment boundaries for a protected execution environment that are typically distributed over several physical platforms. Different instances of several TVDs can co-exist on the same physical platform. Communication within the TVD (intra-TVD communication) is possible through the use of shared resources such as network interface and storage. The underlying Virtual Machine Monitor (VMM) isolates different TVDs and enforces access control to TVD resources according to the underlying TVD policy.

3.5.1 Architecture Overview

Following previously proposed designs [70], our TVD architecture is comprised of two main components. A central component TVD Master is complemented by TVD Proxies; one instance of such a proxy is running on each physical platform hosting a TVD and represents the local copy of TVD Master. TVD Master stores and enforces the corresponding policy TVD Policy for admission of physical platforms, whereas TVD Proxy enforces the policy for admission of VMs. TVD Master is a logical service and could be implemented either on a single server or in distributed way. A TVD Policy defines the security policy of the TVD in the following way:

1. Configurations of virtualization platforms that are trusted to run the TVD. We denote such platforms as TVD Platforms. TVD Platforms configuration is represented by integrity measurements\(^3\) of a platform’s TCB.

2. Configurations of VMs which are trusted to be executed on TVD Platforms and allowed to join the TVD. Such VMs are called TVD VMs. By VM configuration we mean integrity measurement of VM’s binaries.

3. TVD Resources like network, storage or special devices that have to be available to individual TVD VMs or their groups.

4. Intra-TVD access rules describing access restrictions within the TVD.

5. Credentials necessary to establish secure communication channels, e.g., cryptographic keys or certificates of the TVD.

Each platform has one TVD Proxy for each TVD. Before a VM running on TVD Platform can join a TVD, the corresponding TVD Proxy has to be instantiated on the

\(^3\)Here calculated as a cryptographic hash values of the corresponding binaries
platform. During this instantiation, TVD Master deploys TVD Policy to local TVD Proxy. After deployment, TVD Proxy enforces the admission of VMs to the TVD locally on the respective platform. Figure 3.5 shows our TVD architecture (see also [7]).

To securely deploy and locally enforce TVD Policy, TVD Master has to rely on the TCB on each platform. The fundamental building block of the TCB is a virtualization layer that is able to run isolated VMs. The other main TCB components are TVD Proxy Factory, ResourceMgr, CompartmentMgr and TrustMgr.

TVD Proxy Factory is responsible for spawning new TVD Proxy instances. ResourceMgr provides access to basic virtualized resources TVD Resources like networking and storage. In case of TVD VMs, access to these resources is controlled by TVD Proxy. CompartmentMgr is responsible for managing virtual machines (compartments) and their attributes. It starts, stops, pauses VMs and attests their configuration, i.e., performs integrity measurement.

TrustMgr provides an abstraction of the underlying trusted computing functionality in hardware, here the Trusted Platform Module (TPM) [311]. TrustMgr is responsible for generation of cryptographic keys (and corresponding certificates) that are protected by the TPM and are bound to the configuration (integrity measurements) of the components of the underlying TCB. Integrity measurements, keys, and certificates allow a remote party to establish a trusted channel to the platform.

3.5.2 The VDC Demonstrator

We use the TVD concept to realize a Virtual Data Center (VDC) prototype, which was the demonstration platform developed and used in the OpenTC project [230, 7]. In the VDC demonstrator, customers can rent a complete virtual computing infrastructure from a data center operator, to run their own virtual data center therein. Within the resource-constraints of their VDC, the customer can set up one or more TVDs by defining a TVD Policy for each one. Customers can provide their policy definition and manage the TVD through TVD management consoles, which are also part of their respective TVD and run
either on a separate VDC management platform, e.g., on the customer’s laptop.

The high-level architecture of the prototype is illustrated in Figure 3.6. It consists of
three Computing Platforms and one Management Platform. Two Computing Platforms
are located in the data center and another one is connected to data center remotely. We use
two switches to represent two different physical networks: The local network of the data
center and the Internet. Inside the data center, the Management Platform (Platform \#3)
is an accumulation of all servers required for normal operation of the data center, e.g.,
providing services for booting over network or assigning network addresses. Moreover, this
platform realizes basic TVD infrastructure services like the TVD Master. It also provides
the uplink to the Internet for the physical VDC as well as possibly defined TVDs. The
Computing Platforms execute the actual workload, i.e., the TVD VMs. All machines in
the data center are Xen based Linux hosts [46], while the remote Computing Platform
(Platform \#4) is implemented on the L4/Fiasco microkernel [148] and represents a typical
home computer.

The VDC Demonstrator runs two TVDs labeled as \texttt{red.tvd.net} and \texttt{blue.tvd.net}, or
\texttt{red} and \texttt{blue} for short. Each TVD comprises a set of VMs and logical networks that are
specified in its TVD Policy.

The Computing Platform is intended to be the remote administration console for
the TVDs to which it is connected (both, \texttt{blue} and \texttt{red} in our demo). For each TVD,
there is a dedicated management VM running isolated from other VMs on this platform.
Depending on the TVD Policy of each TVD, the management VM allows the local user
of this platform to remotely access other VMs inside the TVD. We provide a graphical
interface and allow the user to manage and access only those VMs that belong to the
corresponding TVD and that the user has access to. The underlying network tunneling
and policy enforcement is fully transparent to the user, who just has to define the (virtual)
networks of his TVD and can then start VMs and join them to the TVD.
3.5.3 Hypervisor Abstraction and Management

For automated remote management of compartments, we use the libvirt virtualization API [194]. Libvirt is a toolkit that features a common interface to use the virtualization functionalities of several mainstream open source virtual machine monitors including Xen [46], KVM [244], QEMU [48], and VirtualBox [291].

We implemented a libvirt interface to the L4 microkernel to allow libvirt to handle L4 in the same way as other supported hypervisors. As a result, we can integrate the L4 systems transparently into the management interface of the VDC and TVD administrators. Furthermore, to meet the security requirements of our project, we extend the libvirt with a Role-Based Access Control (RBAC) module.

The RBAC module enforces the isolation in the TVD management by defining a distinguished role for the administrator of each TVD and by creating a separated view of the VDC resources for each role on a per-TVD basis. These views are defined through a set of rules that filter the access to the different resources according to their “ownership tag” that is the identifier of the TVD they belong to. The ownership tag is initially assigned to the administrator (i.e., it is associated to the corresponding role), and it is propagated to any VM the administrator requests to create and to join to the corresponding TVD.

For the integration in the Xen hypervisor, we additionally implemented a relay module that operates after the RBAC module. It intercepts requests on resources that are owned by the TVD, and lets CompartmenMgr carry out the associated security tasks, such as attestation and connection of the protected TVD Resources (e.g., encrypted storage).

3.5.4 Secure Virtual Networks

In context of VDCs, one can usually assume that the internal VDC network is trusted, while the communication channels to and from the VDC can be attacked. Hence, we use Virtual LAN (VLAN) [150] inside the VDC and labeled IPsec tunnels [158] in other cases.

VLAN-based virtualization provides easy and highly efficient integration into existing Ethernet networks, but it does not provide encryption and cannot defend against eavesdropping. In comparison, IPsec-based network virtualization is less efficient and more complex than VLAN, but does not require a trusted physical network infrastructure and provides much more flexibility by running on top of IP.

We achieve an optimal trade-off between isolation of TVD VMs and remote management access to the TVD by introducing a separate management network for each TVD (see Figure 3.7). The main purpose of this network is to provide limited access to the hypervisor interface on each Computing Platform. This interface allows TVD owners (e.g., VDC customers) to create and manage the virtual machines (TVD VMs) and associated TVD Resources. To remotely manage the TVD, the TVD owner downloads a management VM provided by the TVD infrastructure provider and uses it to join the TVD network. According to TVD Policy, the management VM is automatically associated with the respective networks, in this case the management network. This enables the VM to access the hypervisor interface of all Computing Platforms that the TVD was expanded to.

Our virtual networks can also be used to provide access to other networks or TVDs to realize inter-TVD communication. For general Internet access, this was implemented
3.6 Lessons Learned

During the development of our TVD infrastructure we encountered several more or less severe issues, sometimes leading to radical design changes such as introducing dedicated management networks. In the following, we discuss some of the main insights we gained building a TVD-based virtual infrastructure using trusted computing.

3.6.1 Multiple Logical Networks per TVD

It became clear when designing our prototype that a TVD must support multiple logical networks with different sets of TVD VMs to achieve maximum isolation and yet allow certain privileged VMs to access management interfaces or external networks. Furthermore, customers will expect general Internet connectivity for some of their VMs as well as the ability to isolate some subsets of TVD VMs from others. For example, a large server infrastructure will typically consist of multiple components such as databases, application servers, proxies for load balancing and security enforcement, as well as systems using a virtual bridge on the Internet gateway of the local physical TVD infrastructure. A corresponding virtual TVD network is provided for each TVD and connected to the individual TVD VMs as specified by the TVD Policy. While inter-TVD communication is possible this way, the resulting exchange point is common to all TVDs and the inter-TVD communication is not isolated from other TVDs.
CHAPTER 3. TRUSTED VIRTUAL NETWORKS

for replication and hot failover.

Such infrastructures require access control within their domain to enhance security and prevent unintended interactions. In real data centers, such access control is typically provided through extensive use of VLAN [150] and packet filtering in the network switches. However, such issues have not been discussed in context of TVD infrastructures before. Some prior work suggests employing the TVD concept on multiple layers to control information flow within a TVD [167]. However, even a multi-layer TVD concept provides much less fine-grained access control than a simple network packet filter. For cloud computing services on the other hand, emerging implementations like Amazon's Compute Cloud already support fine-grained access control and out-of-band security management of the system [31].

In the OpenTC VDC Demonstrator, we found that a minimum of two logical networks is needed for each TVD, a user network that includes all user-defined VMs of a TVD and a management network that includes interfaces to a VM management interface. Additional networks should be employed for coarse-grained domain partitioning, and fine-grained access control should be deployed between logical networks and on individual hosts. This combination provides expert customers with the required security features while providing an intuitive security baseline for novices.

3.6.2 Code Re-Use in Secure OS Designs

The OpenTC TVD Policy was designed such that it should be flexible enough to handle future TVD developments and different architectures for measuring components and trusted channels. The resulting relatively complex XML policy structure was simplified in the L4 implementation by automated pre-processing in the Linux-based TVD-Master, allowing the use of a simple plain text parser to be implemented. In a similar vein, basic libraries for string processing and even cryptographic libraries have been reimplemented in previous work [103, 250, 15] with the goal to reduce the TCB and improve trustworthiness of software modules. It is questionable however if this approach is reasonable, since it resulted in a number of ad-hoc reimplementations and partial ports of basic system functions and libraries with unfamiliar behavior.

Instead, the focus of future work should be on re-using well-tested and established software modules and interfaces. If the security of legacy software is questioned, it must be investigated if additional isolation and access control can be provided or if the development of own components is absolutely necessary. Extra consideration must be given to the employed programming languages as to reduce sources of error if the subsystem is not performance critical.

3.6.3 Secure Resource Labeling

Low-level resource labels must be ephemeral and hard to guess, to allow effective exclusion of revoked parties. Further, accidental label collisions between TVDs must be prevented, which are well conceivable when a TVD is deployed to several TVD infrastructure providers at once.
3.7. RELATED WORK

We therefore propose to use low-level labels that are negotiated on demand between TVD members or management components, and used by the TVD infrastructure to efficiently enforce access control between TVD Resources and TVD VMs. If we consider labeled IPsec as a mechanism to implement a TVD Network, this becomes immediately obvious: To secure access to this resource and assure that revoked hosts are reliably excluded, a negotiation is needed between the corresponding Computing Platforms. In this case, the TVD Master will issue a new set of authorization tokens on each policy update to assure that Computing Platforms with outdated or revoked policies cannot participate in these negotiations. For the implementation of the label negotiation, we refer to existing work on group key exchange [205].

3.6.4 Information Leakage and Covert Channels

A major problem of isolating machines and networks in a virtual infrastructure is the inherent use of shared resources and associated side-effects between supposedly “isolated” systems. In particular, an isolation architecture based on virtualization typically assumes that different virtual systems can only communicate through explicitly supplied interfaces, although it is well known and documented that resources shared in “best effort” manner are also a source of potential covert and side channels.

In particular, communication networks, storage pools, and CPU access in shared systems have been used previously to determine if and how the resource is accessed by other virtualized systems, to communicate with those other systems or to infer information about the workload or even application data and cryptographic keys (e.g., [231, 131, 248]).

3.7 Related Work

The TCG work group for Trusted Network Connect (TNC) published several specifications on the integration of remote attestation into existing secure channel protocols. Their proposed TNC architecture [306] is a general framework for request, transmission and validation of attestation reports: Attestation data is exchanged between multiple Agents on the involved network endpoints. The messages are collected from the Agents [304, 305], and encapsulated in the TNC Client Server (TNCCS) signaling protocol [310]. Two alternative protocols are specified to transport these TNCCS messages to the peer, one using the EAP framework [21] and one using a separate dedicated TLS [309] channel. Although the TNC framework provides a unified approach to extending the many existing secure channel protocols, the resulting constructions suffer from substantial complexity as outlined in Section 3.4. As of today, the TNC framework also failed to address the problem of managing and verifying the reported measurements.

Multiple alternative protocols have been proposed as a result of these shortcomings. Similar to the presented extension of IKEv2, the authors of [37] propose an extension of TLS to transmit attestation data payloads. Another approach is to create sealed attestation keys in a trusted environment [41, 42]. This allows establishing an asynchronous trusted channel where sender has assurance that the data can only be accessed when the receiver is in a good state. A first approach to automatically collect and validate
binary code and data measurements of complete computer systems was presented in [6]. However, additional research on software fault containment and life-cycle management of attestation reports seems to be required before these results can be useful for commodity computer systems.

Trusted Virtual Domains (TVDs) were first proposed in [135, 67]. Various applications of TVDs have already been shown and discussed, for instance, applying the TVD concept for secure information sharing [167], enterprise rights management [123], or virtual data centers [52, 69, 51]. In [52] and [69] the authors discuss the management of TVDs in data centers and present a high-level architecture for such a TVD infrastructure. Another interesting extension is the integration of mobile storage devices such as USB sticks [75].

To the best of our knowledge, our implementation of TVDs is the first to span a heterogeneous environment, including detailed proposed protocols for exchanging attestation data, and approach the automated deployment and management of TVDs.

3.8 Summary

We proposed an extension to the IKEv2 key exchange protocol used in IPsec VPNs. By transporting tunneling arbitrary remote attestation messages within the IKEv2 signaling channel, our construction is of much lower complexity than previously proposed designs. The result is particularly interesting for use with resource-constrained devices or if formal verification is desired. As IKEv2 is designed as a generic key exchange server, our solution is also more versatile than previous TLS-based trusted channels.

We have evaluated the concept of trusted VPNs by constructing virtualized networks for the OpenTC Virtual Data Center prototype and presented the “lessons learned” from this experience. In particular, we found that secure virtual networks assume a level of isolation which is not traditionally part of the typical VPN or secure channel adversary model. In relying on the abstract concept of a VPN, the problem of covert channels and traffic analysis may be neglected, leading to unexpected side-effects in such seemingly “isolated” virtual infrastructures. We investigate this in the following Chapters 4 and 5.
Chapter 4

Analysis and Mitigation of VPN Covert Channels

Virtual Private Networks (VPNs) are popular means for enterprises and organizations to securely connect their network sites over the Internet. Their security is implemented and enforced by VPN gateways that tunnel the transferred data in secure channels, thus logically connecting the remote sites in an isolated network. Abstracted this way, VPNs are increasingly used in scenarios that secure channels were not designed for: to logically isolate networks, providing “networks as a service” in virtualized environments like Clouds, Trusted Virtual Domains, or the Future Internet [83, 7, 73]. However, as pointed out in Section 3.6, covert channels and information leakage through shared resources can be a major issue in these new usages, and are still neglected in most designs.

Covert channels violate the system security policy by using channels “not intended for information transfer at all” [190, 219]. While there is a large body of research on covert channels, few works have considered the practical implementation and performance impact of comprehensive covert channel mitigation in modern networks. We believe such work is important for a number of reasons, especially regarding VPNs and network virtualization:

(1) Insider Threat: In contrast to end-to-end secure channels, where the endpoints are implicitly trusted, VPNs are also used for logical network isolation and perimeter security enforcement. In this context, the members of a VPN are often not fully trusted, but instead the trust is reduced to central policy enforcement points, the VPN gateways, which should prevent undesired information flows. However, malicious insiders in the LAN may leak information through the VPN gateways using covert channels, thus circumventing the security policy. Examples of such insiders can be actual humans or stealth malware, engaging in industrial espionage, leaking realtime financial transaction data, or disclosing large amounts of data from physically secured institutions (e.g., to Wikileaks).

(2) Traffic Analysis: By analysing traffic patterns and meta-data, it is also possible to infer information about transferred data without assuming a malicious insider [321, 193]. Such “passive” Man-in-the-Middle (MITM) scenarios are becoming more prevalent with network virtualization, allowing co-located, supposedly isolated systems to analyse each other [250]. To mitigate such attacks, a common approach is to consider the maximum possible information leakage by assuming colluding malicious insiders. By limiting this information leakage, covert channel mitigation thus also affects traffic analysis [133].
CHAPTER 4. ANALYSIS AND MITIGATION OF VPN COVERT CHANNELS

(3) Combination with Detection: Although application-layer firewalls and intrusion detection systems are widely deployed, carefully designed covert channels remain hard to detect [10, 215]. In these systems, the adversary chooses a weaker signal and mimics the patterns of regular channel usage. Covert channel mitigation can be useful here to induce noise, forcing the adversary to use a stronger signal and thus facilitate detection. We expect the combination of covert channel mitigation and detection to allow for less intrusive pattern enforcement and thus significantly reduce the performance penalty.

In this chapter we present the first comprehensive analysis of covert channels in IPSec based VPNs and a set of techniques and mechanisms to mitigate them. We first define the problem of VPN covert channels in Section 4.1. In Section 4.2 we then identify and categorize the different types of covert channels and determine their capacity. We present a framework and several techniques for high-performance covert channel mitigation in Section 4.3. In particular, we propose an algorithm for on-demand adjustment of traffic pattern enforcement in Section 4.3.2 that increases peak network performance while also reducing overhead during reduced usage. We present a practical instantiation of this framework for the Linux IPSec stack and analyse its performance for different kinds of traffic in Section 4.4. In contrast to previous works, which achieve throughput rates in the range of modem speed [319, 133] and taunt the strong performance impact of proposed mitigation mechanisms [212], our prototype achieves 169 Mbit/s in a 200 Mbit/s VPN link, and a significantly lower overall protocol overhead of 24%.

The source code of our prototype and tools are available online: https://code.google.com/p/ipsec-tfc/.

4.1 Problem Setting and Adversary Model

In the following we define the problem of covert channels in VPNs. Note that our definition differs from previous, less explicit considerations, which consider communication between legitimate VPN participants and are better described as steganographic channels [174, 25, 189]. Although we limit ourselves to VPNs in state-of-the-art IPsec configuration [90], our results can be generalized to other IPsec configurations; and our proposed performance optimizations for mitigating timing channels are also applicable to any other packet-oriented secure channels.

4.1.1 System Model and Terminology

As illustrated in Figure 4.1, we consider a VPN comprising two or more Local Area Networks (LANs) that are inter-connected over an insecure Wide Area Network (WAN). In our scenario, the security goal of the VPN is not only to provide a secure channel (confidentiality, authenticity, integrity) but also to confine communication of LAN hosts to the VPN, i.e., to isolate the protected from the unprotected domain. VPNs are increasingly used for such logical isolation, to create secure virtualized or overlay networks, or simply enforce perimeter security in large companies [83, 7, 73]. This de-facto security goal of isolating the protected from the unprotected domain, and its efficient implementation, is the focus of this chapter.
4.1. PROBLEM SETTING AND ADVERSARY MODEL

Figure 4.1: Problem scenario: A VPN with three LAN sites. The adversary aims to exchange information between the MITM and malicious insiders using covert channels.

For this purpose, we distinguish legitimate channels that transfer and protect user data according to the VPN security policy from covert channels that can be used to circumvent this policy. Covert channels exist because the legitimate channel acts as a shared resource between the protected and unprotected domain, exhibiting certain characteristics that can be manipulated and measured by different parties. We denote channels from the protected to unprotected domain and vice versa as outbound and inbound covert channels, respectively.

We measure the security of our system using the Shannon capacity of the covert channels, i.e., the information theoretic limit on the amount of information that can be transferred through them [321]. The covert channel capacity is given in bits per legitimate channel packet (bpp) or, where applicable, in bits per second (bps). The capacity of each covert channel type is denoted as $C_{\text{type}}$. The capacities are classified as maximum ($m$) vs. remaining ($r$) covert channel rate for inbound ($in$) vs. outbound ($out$) covert channels. For example, the maximum capacity of the outbound covert channel based on packet size is denoted as $C_{\text{PktSize},m,out}$, or as $C_{\text{PktSize},r,out}$ after countermeasures have been applied. The remaining aggregated inbound and outbound covert channel rates are denoted as $\hat{C}_{r,in}$ and $\hat{C}_{r,out}$, respectively.

4.1.2 Adversary Model

The adversary controls one or more compromised hosts in the LAN sites as well as an active MITM in the WAN. We refer to the LAN hosts controlled by the adversary as (malicious) insiders, regardless of whether they are controlled by actual humans or stealth malware. The adversary’s goal is to establish a communication channel between the MITM and one or more possibly colluding malicious insiders, as illustrated in Figure 4.1. This would allow the adversary to send instructions to the insiders or to leak information from the protected to the unprotected domain, breaching the perimeter security of the VPN. For this purpose, we assume a state-of-the-art IPsec configuration with authenticated encryption using Encapsulated Security Payload (ESP) in tunnel mode [90], and the cryptographic primitives and keys of the VPN are securely enforced by the VPN gate-
### Table 4.1: Inbound and outbound covert channels capacities for an IPsec VPN with $N+1$ endpoints.

<table>
<thead>
<tr>
<th>Class</th>
<th>Type</th>
<th>Capacity $C_{in}$ in bpp</th>
<th>Outbound</th>
<th>Inbound</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage</td>
<td>ECN</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DS</td>
<td>6</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Flags</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>timing/logic</td>
<td>PktSize</td>
<td>8.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IPD</td>
<td>$\geq 1$</td>
<td></td>
<td>$\geq 1$</td>
</tr>
<tr>
<td></td>
<td>PktOrd</td>
<td>-</td>
<td></td>
<td>$&gt; 6.58$</td>
</tr>
<tr>
<td></td>
<td>PktDrop</td>
<td>-</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>PMTUD</td>
<td>-</td>
<td></td>
<td>5.18</td>
</tr>
<tr>
<td>amplify</td>
<td>DestIP$^2$</td>
<td>$\log_2(N)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ways. However, the legitimate VPN traffic can be manipulated by malicious parties in the protected and unprotected domains to exchange information that “survives” these packet transformations enforced by the VPN gateways.

Unfortunately, no systematic approach is known for identifying network covert channels apart from exhaustive search, and the categorization as storage or timing channels can be ambiguous [219]. We used a comprehensive analysis on the IPsec specification and related work on covert channels in network protocols (cf. Section 4.5), as well as source code analysis and testing\(^1\) to identify potential covert channels in IPsec VPNs. IP-Tunneling and authenticated encryption by the IPsec gateways greatly simplified this problem, as none of the protocol headers that the MITM can read or modify (i.e., the outer IP and ESP header) are directly available to the LAN hosts.

In total, we have identified only eight covert channels. As shown in Table 4.1, the available covert channels comprise three storage-based channels based on fields in the outer IP header (ECN, DS, Flags) and five timing-based covert channels that manipulate Inter-Packet Delay (IPD), packet order (PktOrd), WAN capacity (PktDrop), and Path MTU Discovery (PMTUD). The remaining characteristic of the respective destination LAN of a packet (DestIP) does not constitute a covert channel in its own right but can act as amplification of other covert channels. A detailed discussion of the covert channels is provided in Section 4.2.

We emphasize that some of these channels are implementation dependant, e.g., the treatment of ECN header flags or PMTUD at the VPN gateway, while others (IPD, PktSize, PktOrd) are generic problems faced by all packet-oriented channels. While we are confident to have identified all covert channels, we cannot account for all possible implementations and interpretations of IPsec. In the following, we thus limit our considerations to the identified attack vectors.

\(^1\)Specifically, we examined the IPsec implementations of the current Linux 2.6.32 to 2.6.38 and OpenBSD 4.7 to 4.8 releases.
4.2 Covert Channels in IPsec

In general, for a channel characteristic to be exploited as a covert channel, it must be measurable after transformation by the VPN gateway. An outbound covert channel requires that certain characteristics of packets remain measurable when they are encapsulated and encrypted when traversing from the LAN to the WAN side. Similarly, inbound covert channels require characteristics to survive the decapsulation from the WAN to the LAN.

In this section we review the impact of covert channels on VPNs that apply to the setup described in Section 4.1. Many of the channels are known, however, their impact is different in VPNs and other mitigations are possible. Unfortunately, no systematic approach exists for identifying covert channels besides systematic exhaustive search [219]. The following results are based on such examination of the IPsec specification [174], which standardizes the process of encapsulation and decapsulation at the VPN boundary, and related work. We then examined recent Linux and OpenBSD implementations to validate\(^2\), exposing notable deviations from the specification.

Observe that the following analysis and experiments are limited to IPv4-based IPsec VPNs, which are still in predominant use today. However, the presented results are also directly applicable to IPv6-based IPsec VPNs. The only difference concerning IPsec covert channels is that the IPv4 Flags characteristic was removed from IPv6, and that the theoretic limit for the number of VPN sites in the DestIP characteristic was increased due to the larger 128bit address space. The new IPv6 flow label as well as the various available option headers in IPv6 do not influence the creation of the outer header and thus cannot pose inbound or outbound covert channels as defined in Section 4.1 [174].

4.2.1 Storage Channels

Several storage-based covert channels have been identified in TCP/IP networks [196, 198, 336]. Storage-based covert channels are usually easy to eliminate, but create reliable and efficient channels when left undetected.

**Differentiated Services (DS).** The 6 bit DS field in the IP header is used to signal service requirements, such as realtime VoIP traffic. Hence it may leak information about the type of the transmitted data even if not actively exploited by an insider. The IPsec specification recognizes this problem but only describes countermeasures for inbound covert channels [174]. The proposed countermeasure is to discard the DS field of the outer IP header during defragmentation, unless explicitly configured to do otherwise. Indeed, the examined Linux implementation honors this recommendation, while OpenBSD always discards the outer header’s DS field during decapsulation. Hence we get \( C_{DS,\text{out}}^{\text{in}} = 6 \text{ bpp} \) in both cases (and likely also for most other IPsec implementations) while the optional decapsulation of the DS field opens another inbound channel of the same capacity \( C_{m,\text{in}}^{\text{DS}} = 6 \text{ bpp} \).

\(^2\)We examined IPsec implementations of recent Linux 2.6.36 and OpenBSD 4.8.
Explicit Congestion Notification (ECN). Explicit Congestion Notification (ECN) uses two bits of the IP header to let routers signal network congestion to the endpoints. As such, ECN between WAN routers and LAN endpoints may directly violate the VPN policy. Originally, a “limited functionality” mode was supplied to eliminate such covert channels in IPsec, which effectively precludes the WAN from ECN signaling [245]. However, the current revision of IPsec specifies that the ECN field should be copied during encapsulation, and one bit is copied back to the inner header on decapsulation, if ECN is enabled on the inner IP header [174]. As such, the specification enables covert channels with a capacity of $C_{m,in}^{ECN} = 1$ and $C_{m,out}^{ECN} = 2$ bpp. The ECN treatment was refined and unified in [62]. After discussing the trade-off between covert channels and ECN, an updated scheme is defined with $C_{m,out}^{ECN} = 2$ bpp and $C_{m,in}^{ECN} = 1.5$ bpp. Furthermore, a legacy-compatible mode with $C_{m,out}^{ECN} = 0$ but $C_{m,in}^{ECN} = 1.5$ is specified. The inbound covert channel is deemed insignificant [62].

The examined Linux and OpenBSD systems both implement the original ECN specification [245]. They can be configured to use “limited functionality” mode, eliminating ECN-based covert channels. However, Linux does not drop packets where the outer ECN field was invalidly set to Congestion Experienced (see also Section 4.2.2).

IPv4 Flags (Flags). The IPv4 Flags field consists of one unused bit (Reserved) and two bits for fragmentation, Don’t Fragment (DF) and More Fragments (MF). The IPsec specification discusses the problem of fragmentation at great length, but refers to standard IP-IP encapsulation [238] for the exact treatment of inner and outer header fields. In general, the IPsec gateway must copy the DF bit if set by the sender to enable PMTUD along the route. Otherwise, if the sender allows fragmentation, the IPsec gateway is advised to still use PMTUD but fragment the original packet before encapsulation. If the advice is followed, no outbound covert channel is created by the Flags field. During decapsulation, any WAN fragments must first be reassembled, eliminating any option for the adversary to encode information for the final recipient of the packet.

The examined OpenBSD implementation correctly resets the DF and MF bits on encapsulation and decapsulation, but copies the Reserved bit as a side effect of this normalization. The Linux implementation copies the DF bit by default unless PMTUD has been disabled by configuration. However, in this case a compatibility issue leads to a repeated incrementation of the IP ID field for fragmentable packets. Hence, both implementations leak information about the Flags field settings on the inner header, creating an outbound covert channel with $C_{m,out}^{Flags} = 1$ bpp.

Destination IP (DestIP). The destination IP address can be used as a covert channel [131]. Although the insider in a VPN cannot directly modify the destination address visible in the WAN, the destination IP can be modified indirectly if more than to LAN sites are connected in the VPN, by addressing the packet to one or the other LAN site. However, and assuming that other covert channels are eliminated, the MITM cannot distinguish the (time and size padded) stream of packets for each VPN channel to determine which channel is being preferred by the insider. Hence, the characteristic only acts as an amplifier for an existing outbound covert channel, multiplying the symbol space of the existing covert channel by the number of alternate destination LANs $N$. 
4.2.2 Timing Channels

**Packet Size (PktSize).** Information can also be encoded in the size of packets [131]. In IPsec VPNs, the created covert channel is a limited outbound channel since (1) the MITM cannot covertly modify the packet size and (2) the symbol space is reduced by ESP payload alignment and block cipher padding [173]. To estimate the available symbol space, consider that the maximum and minimum possible packet length is reduced by protocol headers. The remaining possible packet lengths are then partitioned due to ESP padding and alignment. For example, the maximum packet size in standard Ethernet LAN is limited by the 1500 byte Maximum Transmission Unit (MTU), minus 40 bytes inner and outer IP headers and two times $\approx 12$ bytes for the ESP header with cipher block padding and ESP MAC, respectively. Hence $l_{\text{max}} \approx 1500 - 2 \cdot 20 - 2 \cdot 12 = 1436$ bytes and $l_{\text{min}} \approx 2 \cdot 20 + 2 \cdot 12 = 64^3$. Considering the 4 byte alignment of ESP we get $C_{m,\text{out}}^{\text{PktSize}} = \log_2 \left( \frac{1436 - 64}{4} \right) = 8.4 \text{ bpp}$. 

**Inter-Packet Delay (IPD).** Several previous works discuss exploitation and mitigation techniques for covert channels based on the relative delay between packets (IPD) [131, 214, 137, 53, 10]. The characteristic can be used for both, outbound and inbound covert channels and is not significantly affected by IPsec processing. The channel capacity generally depends on the accuracy of the timing measurements on the receiver [214]. However, when approximating it as simple binary channel that either delays a packet or not, it achieves a capacity of $C_{m}^{\text{IPD}} = 1 \text{ bpp}$. Previously, a throughput of 0.98 bpp was reported on an intercontinental Internet connection [10].

**Packet Order (PktOrd).** Information can also be encoded by changing the order of packets [189, 102]. In IPsec VPNs, the MITM cannot distinguish the order of IPsec payloads. However, the MITM may reorder WAN packets to send information to an insider, creating an inbound covert channel. The receiving IPsec gateway can optionally maintain a partial anti-replay window, a bit mask that marks the last $r$ received packets based on the ESP or AH sequence number field. If used, a typical replay window of, e.g., $r = 32$ packets leaves an inbound covert channel capacity of $C_{m,\text{in}}^{\text{PktOrd}} = \frac{1}{r} \cdot \log_2 (r!) = 3.67 \text{ bpp}$. 

**Packet Drops (PktDrop).** Packet dropping was proposed in [270] to create covert channels. However, since the MITM in a VPN cannot distinguish secure channel packets, the outbound version of this channel is equivalent to the IPD-based covert channel. When used as an inbound covert channel, one LAN client may generate a stream of enumerated packets from which the WAN MITM adversary may drop some at will. Hence, a reliable covert channel with at most $C_{m,\text{in}}^{\text{PktDrop}} = 1 \text{ bpp (drop/no drop)}$ can be created.

**Path MTU Discovery (PMTUD).** PMTUD is a mechanism to dynamically detect the Path MTU (PMTU), the maximum allowed packet size along an IP path [213]. By setting the Don’t Fragment (DF) flag in the IP header, the sender asks intermediate

---

3Excluding the block cipher’s initialization vector (IV).
routers to not fragment packets on demand but to drop it and instead report the MTU of their network segment as an ICMP error (PMTU error). The sender notes the respective smallest reported MTU towards a particular destination as the PMTU and adjusts the size of emitted packets accordingly. The sender will also periodically test by trial & error if a larger PMTU is possible, e.g., in case the routing has been changed (PMTU aging).

While the size of VPN packets cannot be manipulated directly, the MITM may artificially limit the MTU of his network segment to inject information via PMTUD. Specifically, when using PMTUD in the WAN, the MITM may inject ICMP errors to reduce the PMTU perceived by the IPsec gateways. Upon receiving a packet that is larger than the known PMTU for the respective WAN destination, and if that packet has the DF bit set, IPsec gateways will propagate the smaller PMTU to the respective LAN client by synthesizing the appropriate PMTU error [174].

Since the PMTU is unlikely to change frequently, the IPsec specification recommends that gateways follow the PMTU aging process described in the PMTUD specification [213]. There, a periodic timeout of \( t = 2 \) minutes is recommended before attempting to increase the PMTU, and immediately increasing further if the attempt was successful, until the new PMTU is discovered. Since only smaller PMTU values are propagated to LAN clients, this limits the rate of PMTU errors that the MTIM can usefully inject.

For example, if we assume possible MTU range from the minimum IP packet size (768 bytes) to the maximum Ethernet MTU minus IPsec overhead (\( \approx 1500 - 20 - 35 = 1445 \) bytes), with changes in 4 byte steps due to ESP padding and alignment, there are \( M \approx \frac{1445 - 768}{4} \approx 169 \) different packet sizes that could be reported by the IPsec gateway to the LAN client. Furthermore, the adversary can mitigate the delay imposed by the PMTU aging timeout \( t \) by partitioning the symbol space such that at most \( \frac{M}{2} = 169/2 = 84 \) subsequent 1 bit symbols can be sent. As a result, up to \( C_{m,in}^{\text{PMTUD}} = \frac{M/2}{t} = 84 \cdot 60 = 5.18 \) bps can be achieved.

### 4.2.3 Active Probing

The MITM may also infer information on the LAN status by actively probing the IPsec gateways and evaluating their response behavior, an approach that was introduced as active traffic analysis [114, 115]. In particular, a LAN client could cause high load on the LAN interface of the IPsec gateway. The resulting change in the gateway’s system load can then be measured by how it responds to legitimate service requests by the MITM, such as Internet Control Message Protocol (ICMP) ping requests [114].

Note that, contrary to the previous channel characteristics, this attack actually exploits a side channel at the gateway: Its capacity does not depend on the usage of the VPN channel but on the frequency at which the insider can induce high and low system loads at the gateway as well as on the rate at which the MITM is able to probe the gateway to measure its system load with sufficient accuracy. It was previously proposed to either normalize the (seemingly uncritical) responses by the gateway. We also believe that the attack can be prevented by combining the IPsec gateway with a second physical firewall on the WAN side, to filter invalid requests and act as key negotiation server on behalf of the VPN gateway. However, such side-channels are outside the scope of this work.
4.3 Covert Channel-Resilient IPsec

In this section we present the design of a high-performance covert channel-resilient IPsec, i.e., a system with low, known covert channel capacity and high throughput. We present novel or improved techniques for efficient covert channel mitigation in Section 4.3.1. Section 4.3.2 considers the performance of different mitigation strategies, introducing on-demand performance trade-offs. Finally, we derive the remaining aggregated inbound and outbound covert channel capacities of the system in Section 4.3.3.

4.3.1 Covert Channel Mitigation

In the following we present and improve efficient mitigation mechanisms for each of the covert channels identified in Section 4.2.

Packet Size (PktSize)

The packet size characteristic is usually addressed by padding packets to maximum size or assuming them to be of constant size [321]. However, as the product $\text{throughput} = \text{pkt\_size} \cdot \text{pkt\_rate}$ is constant for a given link, enforcement of small packet sizes can reduce the load per packet significantly, allowing higher packet rates and more simultaneous connections.

It was previously proposed to allow multiple alternate packet sizes [131], but then the ratio between packets of different sizes creates another covert channel. Mode Security [63] was proposed to manage the switching between different enforcement modes and audit such a remaining covert channel. However, real network traffic is often mixed, i.e., packet streams using different packet sizes are often transmitted at the same time. Moreover, the enforcement of small packet sizes is problematic for IP protocols: With Path MTU Discovery (PMTUD), the connection endpoints quickly detect and adapt to the maximum allowed packet size of an IP route, but only slowly recover to a larger MTU using a conservative trial-and-error approach. This active adaption also makes it harder for the VPN gateways to estimate the actual demand for packets of larger size.

We address these problems by combining packet padding with transparent fragmentation and multiplexing, mechanisms that were previously only considered for traffic pattern obfuscation [176, 330]. Packet fragmentation within IPsec allows us to efficiently and transparently enforce various packet sizes at the gateway without influencing the channel’s Path MTU (PMTU). This is different from regular IP fragmentation before or after IPsec processing, which results in visible fragments either on the LAN or WAN sides that could again be used as covert channels. The fragmentation mechanism is complemented by packet multiplexing, which can be used to reduce packet padding overhead by concatenating multiple smaller packets up to the desired packet size. This also reduces the IPsec encapsulation overhead (ESP, IP).

When working with mixed traffic, the sender gateway first fragments large packets and then attempts to multiplex small packets or fragments into the padding area of previously processed packets that are still in the packet buffer. At the receiving gateway, packets are first de-multiplexed and then defragmented. As this mechanism works transparently
for the LAN sender and receiver, the LAN gateways can precisely monitor the current demands of the adjacent LAN site to optimally adjust the enforced packet size.

**Inter-Packet Delay (IPD)**

The covert channel based on IPDs and its mitigation were subject of several previous works (e.g., [214, 114, 321, 320, 10]). In theory, it is easily eliminated by enforcing a fixed IPD at the VPN gateway, inserting dummy packets when no real packets are available [320]. However, due to the very high packet rates in modern networks, even short periods of non-optimal IPDs (and thus packet rate) enforcement at the VPN gateway quickly result in packet loss due to buffer overflows or network congestion. This is often aggravated by the deployed congestion avoidance algorithms, which will quickly adapt to a lower enforced packet rate while only slowly making use of increased rates, thus making it difficult for the VPN gateway to estimate the optimal enforcement rate. The effect can be partly mitigated with larger packet buffers; however, this can also create high packet delays, degrading network responsiveness [127]. Further, the optimal enforced packet rate can be very large in modern networks, creating a high computational overhead for the time-synchronous packet processing. For example, to saturate a 100 Mbit/s link with 200 byte packets, an average IPD of \( \frac{500 \text{ byte}}{100 \cdot 10^6 \text{ byte/s}} = 2 \mu s \) should be enforced. Finally, one must consider inaccuracies in the timing enforcement that appear at high system loads [114, 116]: Since high activity on the LAN interface can influence the system load of the gateway, a LAN host may induce inaccuracies in the IPD enforcement of the gateway that can again be measured by the Man-in-the-Middle (MITM), yielding \( C_{r \text{IPD}} = 0.16 \text{ bps} \) [133].

We have implemented traffic reshaping inside the Linux kernel, using the kernel's High-Precision Event Timer (HPET) infrastructure for packet scheduling with nanosecond resolution. This substantially reduces the overhead of context switching and buffering, allowing IPDs in the range of microseconds rather than several milliseconds (e.g., [319, 133]) and noticeably improves throughput and responsiveness. To maintain good system performance at even higher packet rates we use packet bursts, i.e., we translate very low IPDs into bursts of multiple packets at correspondingly larger delays. For optimal packet buffering we adjust the buffer size depending on the currently enforced IPD. This prevents long delays at low rates while allowing generous buffering at high rates.

Additionally, to address the problem of timing inaccuracies, we use the high resolution of the HPET timers to monitor and actively compensate for timing inaccuracies in randomized IPD enforcement. Specifically, we exploit the fact that determining timing enforcement inaccuracies is harder for the remote MITM than on the local system. The adversary always requires significantly more measurements to first detect the variance of the random IPD enforcement and then the inaccuracy in the enforced variance [114], while the VPN gateway itself can directly compare the intended versus actual packet sending time. Hence, the gateway can approximate the current inaccuracy much faster. Given this knowledge of unintended change in IPD variance, we let the VPN gateway compensate for the enforcement inaccuracy by dynamically adapting the variance of the randomized IPD enforcement. This prevents the adversary from ever collecting sufficient measurement samples for a particular IPD enforcement variance, preventing it from estimating the enforcement inaccuracy and eliminating the timing channel \( C_{r \text{IPD}} = 0 \). However,
further evaluation with specialized network diagnosis hardware is needed to confirm (the non-existence of) this effect.

Packet Order (PktOrd)

Sequence numbers in protocol headers have been used before to create a covert or steganographic channel based on packet reordering [189, 102]. However, in contrast to previous works we can eliminate this channel in the VPN scenario using the IPsec anti-replay window and secure sequence numbers in Encapsulated Security Payload (ESP).

IPsec implementations already maintain a bitmap of the last \( r \) seen and unseen sequence numbers so that replay attacks within the window size can be detected. To eliminate communication through packet re-ordering, we propose to implement this window as a packet buffer, where new packets are inserted \( \text{sorted} \) by their ESP sequence number. When an authenticated packet with a higher sequence number than the currently considered set of \( r \) is observed, the window advances to that highest seen number and any packets falling out of the buffer are forwarded. As a result, all packets forwarded from the VPN gateway into the LAN are ordered, regardless of packet drops, and the covert channel is eliminated: \( C_{r,in}^{\text{PktOrd}} = 0 \).

Unfortunately, the approach is problematic for low packet rates, since the window may advance slowly and individual packets are not forwarded fast enough. We solve this by establishing a certain maximum IPD (e.g., 50ms) and assuring that at least \( r \) dummy packets are sent by a gateway before a connection is stopped. These constraints are necessary in any case to assure network responsiveness and hide short periods of inactivity.

Packet Drops (PktDrop)

In general, it appears impossible to eliminate covert channels based on packet dropping in the WAN. Mitigation with error correction codes is expensive and easily defeated by dropping even more packets. Instead, we propose to mitigate the channel by injecting noise, by \( \text{increasing} \) packet loss proportionally to the actual packet loss.

Specifically, let the gateways maintain a buffer \( D \) of size \( l \). At the sender gateway, packets are buffered in \( D \) and their order is randomized before encapsulation. At the receiver gateway, the packets are again collected in \( D \) and the number of dropped packets \( i \) in a sequence of \( l \) packets is determined based on their ESP sequence number. If \( i > 0 \), the gateway drops another \( j \) packets from the current buffer, such that \( i + j = 2^x \), for \( 1 < x \leq \log_2(l) \), and forwards the remaining packets after randomizing their order once again. As a result, the MITM can choose the overall number of packets to be dropped for the receiving LAN client but cannot select which packets to drop, resulting in a symbol space of \( \log_2(l) + 1 \) packets per \( l \) packets. The remaining covert channel capacity is then \( C_{r,in}^{\text{PktDrop}} = \frac{1}{l} \cdot \log_2(\log_2(l) + 1) \text{ bpp} \).

As in the packet re-ordering mitigation in Section 4.3.1, the inbound packet buffer \( D \) at the receiving gateway is problematic for very low packet rates and requires similar restrictions to assure a steady stream of packets.

Note that implementations could combine the inbound packet sorting and IPD enforcement with the packet dropping facility to mitigate the delays of using multiple queues.
Path MTU Discovery (PMTUD)

To our knowledge, no previous work considered the possibility of covert channels based on PMTUD, in particular with respect to VPNs. Since PMTUD is critical for good network performance, we do not disable it but instead mitigate the channel by enforcing limits on the rate and values that are propagated by the VPN gateways into the LAN.

In particular, we limit the possible PMTU values by maintaining a list of common PMTU values and only propagate the respective next lower PMTU to the LAN. Such common PMTUs values can be established on site or can be derived from previously proposed performance optimizations for PMTUD [213]. The rate limitation of PMTU propagation is problematic in general, as a lack of MTU adaption will lead to packet loss. However, in our case the current PMTU is always known to the trusted VPN gateways, which can then use the transparent fragmentation feature from PktSize enforcement to translate between LAN and WAN packet sizes. Considering the 10 most common PMTU values and an average interval of, e.g., 2 minutes [213] between propagation of PMTU changes, our measures reduce the covert channel rate to less than $C_{PMTUD} = 0.03 \text{ bps}$.

Storage-based Channels (ECN, DS, Flags)

The storage-based covert channels exploiting the Explicit Congestion Notification (ECN), Differentiated Services (DS) and IPv4 Flags handling of IP/IPsec are easily eliminated by resetting the respective fields of the outer IP header at encapsulation and ignoring them during decapsulation. Normalizing the IPv4 Flags field is unproblematic as en-route fragmentation is deprecated in IP. However, eliminating the ECN and DS covert channels disables these performance optimizations in the WAN.

4.3.2 Mitigation Policies and Performance

In this section, we discuss different covert channel mitigation policies that can be enforced using the techniques described in Section 4.3.1. We start by discussing the problems of previously proposed Fully Padded Channel and Mode Security approaches, and then propose a new system for on-demand, dynamic adaption of the enforced channel characteristics. We focus on the IPD and PktSize enforcement policies, as they have by far the highest performance impact.

Fully Padded Channel

When applied without any performance trade-offs, the mitigation mechanisms described in Section 4.3.1 result in a fully padded channel: The WAN packet stream is constantly padded to the maximum desired throughput rate and packet size. However, this mitigation policy has several disadvantages: (1) The system must compromise between high throughput and responsiveness, likely opting to enforce maximum packet sizes to reduce fragmentation overhead; (2) the maximum (desired) network load is constantly enforced in both directions, reducing overall performance due to network congestion; (3) TCP/IP congestion avoidance algorithms do not work, since any rate throttling is compensated by additional channel padding. In case of temporary reductions in WAN capacity, this
4.3. **COVERT CHANNEL-RESILIENT IPSEC**

leads to repeated packet loss and throttling, until the network is not usable anymore. Hence, the *fully* padded channel seems unfit for practical use, except in private/dedicated networks.

**Mode Security**

Mode Security is a generic scheme for trading covert channel-resilience against system performance. This is done by organizing system operation in a set of alternative *operation modes* that can be switched at a certain rate [63]. The current operation mode should be selected such that performance penalty and/or overhead produced by the covert channel mitigation is minimized. Since the enforced operation mode will usually depend on the actually required usage, this adaption itself may be exploited as a covert channel. In this case, the covert channel capacity can be given as $C_{\text{ModeSec}}^{\text{out}} = R \cdot \log_2(M)$, where $M$ is the number of operation modes and $R$ is the maximum rate at which the operation mode can be changed (transition rate).

Mode Security was used to estimate the theoretic network overhead and covert channel capacity [321]. However, this assumes an algorithm that can determine the optimal operation mode at any time. To the best of our knowledge, no practical implementation and evaluation of this mechanism exists; in particular, no strategies have been proposed to automatically determine and apply the optimal operation mode in the face of often unpredictable network traffic. Indeed, our attempts to directly apply Mode Security by determining the best next mode in each interval $R$ resulted in poor performance, as its strategy failed to accommodate the deployed congestion avoidance algorithms. Hence, the following section augments Mode Security with a predictive and flexible management and trade-off scheme.

**On-Demand Mode Security Management**

An algorithm for on-demand adaptation in network covert channel mitigation must accommodate multiple conflicting constraints. It must quickly react to changes in channel usage to elude congestion avoidance algorithms, yet the amount of possible mode changes should be minimal. Moreover, the employed packet queue should buffer packet bursts at various average packet rates, yet react quickly when the current average rate is overused by dropping individual packets. We address these conflicts using the following regulation mechanisms:

**Token Bucket Filter.** We generalize the transition rate $R$ of the Mode Security paradigm to a token bucket filter [340]. Tokens are generated at a fixed rate $R$ and each mode transition consumes a token from the token bucket. This allows us to “save up” unused mode transitions in form of tokens and consume them on demand, at temporarily higher rates than $R$. The amount of cached tokens is limited by the token bucket size and the *average transition rate* $\bar{R}$ is bound by the rate $R$ at which new tokens are generated. Thus, the token bucket filter allows us to immediately react to changes in network usage, before connection throttling kicks in or network delays become noticeable. Further, the token bucket status may influence and optimize decisions on the operation mode to be enforced.
Algorithm 1: Pseudo-code for packet rate adjustment.

```plaintext
while true do
    \((r_{\text{LAN}}, r_{\text{frag}}, r_{\text{mplex}}) \leftarrow \text{get-stats()}
    r_{\text{opt}} \leftarrow r_{\text{LAN}} + r_{\text{frag}} - r_{\text{mplex}}
    r_{\text{avg}} \leftarrow 0.1 \cdot r_{\text{opt}} + 0.9 \cdot r_{\text{avg}}
    \text{case } r_{\text{opt}} > 0.9 \cdot r_{\text{now}} \text{ do}
        r_{\text{amp}} \leftarrow (r_{\text{max}} - r_{\text{opt}})/t_{\text{num}}
        r_{\text{new}} \leftarrow r_{\text{opt}} + \frac{1}{2}r_{\text{quant}} + r_{\text{amp}}
    \text{end}
    \text{case } r_{\text{now}} > 1.1 \cdot r_{\text{avg}} \land t_{\text{num}} > t_{\text{dec}} \text{ do}
        r_{\text{new}} \leftarrow r_{\text{avg}}
    \text{end}
    r_{\text{new}} \leftarrow \text{quantitize}(r_{\text{new}}, r_{\text{quant}})
    \text{sleee}(ival)
end
```

**Aggressive Increase.** Network throughput is scaled mainly based on its packet rate \( r \), with typically exponential rate increase until the first network bottleneck is detected. While the current optimum WAN packet rate \( r_{\text{opt}} \) is easily calculated based on the currently observed LAN rate \( r_{\text{LAN}} \), fragmented and multiplexed packets \( (r_{\text{frag}}, r_{\text{mplex}}) \), the derivation of the next enforced packet rate \( r_{\text{new}} \) is more involved. Algorithm 1 shows a simplified version of our implementation.

To adequately consider exponential rate increases without requiring too frequent changes to \( r_{\text{now}} \), our rate increase phase is designed to constantly overestimate the current optimal packet rate \( r_{\text{opt}} \), by increasing \( r_{\text{now}} \) as soon as it is approached by \( r_{\text{opt}} \) (see Line 5 in Algorithm 1). Combined with buffering and short monitoring intervals \( ival \approx 200\text{ms} \), this approach successfully eludes congestion avoidance algorithms and prevents undesired throughput throttling (constants were derived using empirical tests). However, the overestimation should also not be too large, as it directly affects the padding overhead and can also reduce the inbound traffic rate due to the imposed network load. Moreover, all stored tokens may be used up before a reasonably high packet rate \( r_{\text{now}} \approx r_{\text{max}} \) is reached, resulting in bad performance until new tokens are generated. Hence we also include an amplification mechanism that increases the rate \( r_{\text{opt}} \) in larger steps \( r_{\text{amp}} \), depending on the currently available amount of tokens \( t_{\text{num}} \) (Line 6f.). This prevents the system from becoming “stuck” at low packet rates, at the cost of potentially high padding overhead in cases where such amplification was not required.

**Conservative Slowdown.** When putting the WAN channel in a state of decreased performance, we must take care that sufficient transition tokens are available to adequately adapt to a possible subsequent usage increase as outlined above. In contrast to the aggressive rate increase policy, any reduction in the enforced traffic rate is therefore delayed until a certain amount of tokens \( t_{\text{dec}} \) have been collected in the token bucket. Moreover, to reduce the impact of short-term fluctuations in the packet rate, the rate is only reduced based on the longer-time average traffic rate \( r_{\text{avg}} \), as shown in Algorithm 1 Line 8f. Over-
Dynamic Queue Size with RED. When dynamically adjusting the overall throughput of the WAN channel, we must also adjust the size of the packet queue accordingly. A large queue will incur large delays and timeouts at very low rates, while a small queue is not effective at supporting a channel with very high packet rates. Hence, we dynamically adapt the queue size based on the desired maximum buffering delay and the currently enforced packet rate. Eventually, the WAN channel or its enforcement policy may also reach a point where further rate increases are not possible. In this case, the endpoints should be notified of the increasing congestion quickly, to avoid dropping several packets at once due to full buffers. We realize this by employing Random Early Detection (RED) [58] as the packet queue’s dropping policy, so that endpoints are quickly notified of congestion.

We implemented several variations of this approach and evaluated the effect of different parameters on the short-term and long-term usage adaption. The achieved performance and adaptation behavior is presented in Section 4.4.3.

### 4.3.3 Remaining Covert Channel Capacity

In the following we summarize the identified covert channels and derive the aggregated remaining covert channel capacity of our covert channel-resilient VPN.

Unfortunately, it is not possible to give all the covert channel rates in a closed form and with comparable units. Several covert channels also depend on additional parameters like network PMTU or minimum WAN packet rate. To provide a reasonable overview of the overall effectiveness of the covert channel mitigation, we have used the capacity estimations derived in the examples of Section 4.3.1, assuming a state-of-the-art IPsec VPN configuration (cf. Section 4.1).

Table 4.2 lists the individual covert channel capacities for the unmitigated ($C_m$) and
mitigated ($C_r$) case. Considering that today's networks easily transmit several thousand packets per second, i.e., 1 bpp $\gg$ 1 bps, our system results in significant improvements over standard IPsec. In fact, all outbound covert channels are completely eliminated, except for the DestIP channel. However, as explained in Section 4.2.1, the DestIP characteristic does not by itself constitute a covert channel but can only be used to amplify other channels. Hence, the overall remaining covert channel capacity is given by $\hat{C}_{r,\text{out}} = C_{\text{ModeSec}} \cdot C_{\text{DestIP}}$.

For the less critical inbound covert channels (e.g., control channels for stealth malware), only the channels based on PMTUD and PktDrop remain. The PktDrop covert channel has the highest impact with $C_{r,\text{in}}^{\text{PktDrop}} \leq 5$ bps and is easy to exploit. Since the PMTUD channel could be exploited at the same time, their capacities must be added up: $\hat{C}_{r,\text{in}} = C_{r,\text{in}}^{\text{PktDrop}} + C_{r,\text{in}}^{\text{PMTUD}} = 5.03$ bps.

4.4 Practical Covert Channel Mitigation

In this section we describe the instantiation of our system based on the Linux IPsec stack and analyse the achieved network performance and behavior.

In our prototype implementation and evaluation we only consider the mitigation of outbound covert channels, since information leakage from the protected to the unprotected domain is usually considered more critical (e.g., consider Bell-LaPadula [47]). Moreover, from our discussions in Section 4.3 it is clear that outbound covert channel mitigation is more efficient, as it requires less buffering and processing but is more effective in reducing the covert channel capacity. With these limitations in mind, we believe that the mitigation of inbound covert channels is equally possible in our prototype. We have validated the mitigation of the respective channels by looking at the resulting traffic dumps and using the tools we have developed previously in context of [10].

4.4.1 Architecture and Implementation Details

We have implemented our design as an extension to the IPsec stack of the Linux kernel, called High-Performance Covert Channel Mitigation (HPCM). The implementation is based on the Traffic Flow Confidentiality (TFC) project, a system for probabilistic traffic flow obfuscation and re-routing in IPsec [176]. We revised and extended TFC to support High-Precision Event Timers (HPETs), fragmentation, multiplexing, dummy packet generation that is indistinguishable from real traffic payloads, elimination of storage-based covert channels in the encapsulation headers and, most importantly, an interface for monitoring packet processing statistics and flexible configuration of the traffic pattern enforcement via userspace. The resulting architecture is illustrated in Figure 4.2. In kernelspace, the HPCM Engine processes packets as part of the IPsec subsystem, rewriting problematic header fields and enforcing the currently desired size and IPD constraints as described in Section 4.3.1. In userspace, the HPCM Manager collects processing statistics from the enforcement engine and combines them with the observed inbound LAN traffic to determine the optimal enforcement parameters, as presented in Section 4.3.2. As such, all performance-critical traffic enforcement is performed in kernelspace, while the more volatile security policy management and configuration is flexibly performed in userspace.
4.4. PRACTICAL COVERT CHANNEL MITIGATION

Figure 4.2: Architecture of our Linux prototype.

Packet processing

Packet resizing is done on a best-effort basis to minimize processing delays. Packets that are too large are iteratively fragmented until the last remaining fragment is smaller or equal than the remaining packet size.

Packet multiplexing is performed based on two configurable thresholds, the flagging and the multiplexing threshold. Packets smaller than the flagging threshold are deemed to contain relatively large amounts of padding and are flagged as candidates for multiplexing before they are put into the outbound packet queue. Packets smaller than the flagging threshold are first considered for multiplexing, by searching the current packet queue for previously flagged packets with sufficient padding space and merging the current packet into such a previously processed packet. Only if no suitable candidate can be found in the outbound queue, packets smaller than the multiplexing threshold follow the regular size padding path.

This approach ensures that packet multiplexing has no adverse effect on the packet processing delays and avoids keeping extra state and timeouts for multiplexing candidates. In particular, the approach avoids cases where candidate packets for multiplexing are held up in a separate queue while the packet queue is empty, which would result in sending of dummy packets and negate any performance gained through packet multiplexing.

Since the resulting multiplexed packets typically contain a recent as well as an older packet that was temporarily exempted from the (concurrently running) packet sending process, they are inserted at the start of the packet queue to facilitate their fast sending.

We use the Linux sysfs filesystem to configure the kernelspace HPCM Engine and report real-time processing statistics back to userspace. In particular, the HPCM Engine exports counters for fragmented, padded and multiplexed packets as well as the amount of sent real and dummy packets. The counters can be periodically reset from userspace to yield average processing rates.

Protocol Format

For flexible packet padding and rerouting, we deploy our own encapsulation protocol based on TFC [176] as shown in Figure 4.3. While the length field is sufficient to recognize and remove padding, we require two additional flags to mark packets as using fragmentation
or multiplexing. TFC payloads are flagged as multiplexed if they are followed by another payload, and TFC payloads containing fragments are flagged as fragmented. Fragments payloads are accompanied with a 4 byte fragmentation extension header compatible with IPv4, which allows us to reuse the existing IP defragmentation support in Linux.

The resulting protocol overhead is relatively large, especially due to the additional Security Parameter Index (SPI) field which is used in IPsec to associate the stateless packet stream with some previously negotiated state at the VPN gateways, such as the encryption and authentication algorithms to be employed. However, in case of outbound traffic normalization such stateful processing at the receiver is not actually required. A more optimized implementation could integrate the encapsulation protocol into Encapsulated Security Payload (ESP), requiring only the two flags for marking fragmented and/or multiplexed packets and the optional 4 byte fragmentation extension header.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>No Padding</th>
<th>Fully Padded</th>
<th>On-Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAN Throughput (Mbit/s)</td>
<td>570</td>
<td>201</td>
<td>175</td>
</tr>
<tr>
<td>TCP Transaction Rate (Hz)</td>
<td>1756</td>
<td>1462</td>
<td>1364</td>
</tr>
<tr>
<td>LAN/WAN Overhead (%)</td>
<td>0</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>Relative Throughput (%)</td>
<td>283</td>
<td>100</td>
<td>87</td>
</tr>
</tbody>
</table>

Table 4.3: Throughput and transaction rate for regular and modified IPsec VPN.

4.4.2 Testbed and Raw Performance

In this section we describe the performance achieved by our prototype in terms of network throughput, transaction rate (i.e., roundtrip time) and protocol overhead. Our testbed corresponds to the VPN scenario in Figure 4.1, except that we use only two LAN sites with one physical host per LAN. The Man-in-the-Middle (MITM) is implemented as an Ethernet bridge between the two VPN gateways, allowing reliable observation of all transmitted packets. For our evaluation, the MITM is completely passive and only used to provide independent performance measurements of the WAN. All hosts are 3.2 Ghz Intel Core i5-650 machines, equipped with two Intel PCIe GBit network cards and 4GB system memory. All network links are established at full-duplex GBit/s speed.

\[^4\text{Note that the ESP specification already discusses facilities for traffic normalization but only supports basic packet padding and considers fully padded channels as too costly.}\]
4.4. PRACTICAL COVERT CHANNEL MITIGATION

We have used the Netperf benchmarks TCP_STREAM and TCP_RR to measure the maximum TCP throughput and transaction rate between the LAN sites. By comparing LAN and WAN throughput, we can determine the protocol overhead of the covert channel mitigation, including dummy packets and packet padding.

We list the overall performance results in Table 4.3. The first two columns show the testbed performance for raw IP (plain-text) transmission and IPsec ESP tunneling. With 570 Mbit/s, the raw transmission does not reach the expected GBit throughput, likely due to deficient hardware or drivers. As the LAN hosts and the MITM measure the same IP payloads, there is no LAN/WAN overhead. With 201 Mbit/s, the throughput of a standard IPsec ESP tunnel is already notably slower due to 10% protocol overhead but mainly computational constraints of the VPN gateways. As our covert channel mitigation is an extension of this ESP tunnel setup, we normalize its relative throughput to 100%.

For reference and confirmation of the expected implementation overhead of our prototype, we next evaluated the raw performance of our HPCM Engine compared to the standard IPsec ESP tunnel. The third column “TFC” of Table 4.3 lists the achieved network performance when tunneling TFC inside ESP with all covert channel mitigation techniques disabled. The overall LAN/WAN overhead of 13% (or 3% when compared with the ESP tunnel) is the result of the 8 to 12 byte TFC protocol encapsulation plus some computational overhead.

4.4.3 Covert Channel Mitigation Performance

We now describe the behavior and performance of different mitigation policies. The fourth and fifth column of Table 4.3 show the performance of a “fully padded channel”, enforcing packet sizes of 1422 and 800 bytes at the maximum possible packet rate. For this purpose, we first measured the maximum bi-directional throughput of the VPN channel (201 Mbit/s per direction) and then selected the desired packet rate (inverse IPD) such that the bi-directional channel capacity is almost saturated. We then again measured the maximum (uni-directional) throughput and roundtrip time. As shown in Table 4.3, the fully padded channel configuration achieves rather poor performance in both configurations, reaching only 37% and 28% of the ESP tunnel throughput. Observe that the enforcement of 800 byte packet size achieves higher transaction rate as well as higher throughput. We believe this is due to the overhead of padding TCP acknowledgements to maximum packet size.

We have also implemented and tested an instantiation of our on-demand mode security management scheme presented in Section 4.3.2. As shown in the last column of Table 4.3, the employed mode adaption heuristics reach almost the same throughput as the raw TFC encapsulation without time/size padding (169 Mbit/s vs. 175 Mbit/s), which is the theoretical maximum for our testbed. The LAN/WAN overhead is slightly higher (24% vs. 13%) and the transaction rate rather low. However, we achieve good maximum throughput despite the higher average overhead (84% vs. 87%). This is because, as shown in Figure 4.4(a), the WAN channel experiences overhead mainly in the rate increase and especially rate decrease phases, i.e., when the connection is not actually used much.

5http://www.netperf.org

6As explained in Section 4.3.2, it is critical that the link is not fully saturated since congestion leads to packet loss and congestion avoidance does not work.
(a) WAN adaptation to repeated TCP loads (three consecutive loads shown).

(b) HTTP request delay in mixed traffic.

(c) WAN adaptation to pseudo-random web traffic and downloads.

Figure 4.4: Padding adaptation performance and associated timing/load overheads.

This overhead is the direct result of our design decision in Section 4.3.2 to reduce the frequency of mode adaptations $\bar{R}$ in areas that do not primarily impact user experience. As confirmed by Figure 4.4(a), further reductions of the token regeneration rate $\bar{R} \leq 15^{-1}s$ mainly increases overhead in between TCP loads, without affecting the performance at higher loads or depleting available transition tokens.

Finally, we have investigated the ability of our on-demand mode security management to adapt to random, highly heterogeneous traffic patterns one would expect from a VPN with many users. We used Tsung, a traffic load testing tool\(^7\), to record several HTTP sessions in our network, partly also including larger ($\approx 60$ MB) HTTP downloads. We then configured one of our testbed LANs to act as Internet gateway for the other LAN and used Tsung to replay the recorded HTTP sessions in a pseudo-random fashion with 60 to 80 simultaneous users. Figure 4.4(c) shows how the WAN traffic enforcement for four different token regeneration rates $\bar{R}$ dynamically adapts to the LAN usage (grey filled). For $\bar{R} \leq 15^{-1}s$, only the larger peaks in LAN usage influence the WAN traffic enforcement, reducing information leakage at the cost of padding overhead. As shown in Figure 4.4(b), the mean duration of responding to individual HTTP requests is kept within reasonable limits. However, in contrast to unpadded traffic (grey filled) the accumulated request delays become noticeable to the user.

In the presented configuration, our mode adaptation algorithm switches packet sizes in steps of 100 bytes and packet rates in steps of 1000 packets per second. Considering

\(^7\)http://tsung.erlang-proj-ects.org
4.5 Related Work

Covert Channel Research. There is a large body of literature and history on covert channel and traffic analysis research. For a general introduction we refer to historical discussions in [219, 212] as well as available surveys [196, 198, 336].

In general, covert channels can be classified as timing or storage channels. In covert storage channels, the sender modifies unused or seemingly random bits in the packet to transmit messages [139, 252, 128]. However, many covert storage channels turned out to be easily detectable [215]. In contrast, covert timing channels modulate the message into temporal properties of the traffic.

Instead of using the contents of packets, they convey information through the arrival pattern of packets at the receiver, such as inter-packet delays [233, 131, 68, 276]. Several methods have been proposed to detect or disrupt covert timing channels. Detection primarily uses statistical tests to distinguish covert from legitimate traffic, as the modulation of timing patterns typically results in distinctive timing characteristics. Some earlier works have used anomaly detection to detect and re-shape undesired traffic patterns [298, 299, 297] while more recent works use statistical tests that examine the shape and regularity of traffic [53, 68]. Unfortunately, such detection schemes are rendered ineffective in face of targeted covert channel exploitation [10, 337], and it is hard to provide security guarantees based on such probabilistic obfuscation.

More recent works focus on efficient traffic analysis resilience, mostly based on probabilistic traffic obfuscation schemes such as HTTPoS [199] Traffic Morphing [330] or TFC [176]. However, common problem of these approaches is that it is hard to measure the required level of traffic pattern obfuscation, as evidenced in by several successful attacks based on machine learning algorithms [98].

Efficient Mitigation and IPsec. Although several works consider the problem of covert channels and covert channel mitigation in the TCP/IP protocols [196, 336], we know of no works that specially discuss the problem of covert channels in IPsec. The covert channels we identify in IPsec are generally known, but we found no previous discussion of the PMTUD channel. Additionally, the PktSize [131], PktSort [215, 102] and DestIP [131] characteristics have different impact in IPsec, and the discussion of storage-based covert channels in the IPsec specification [174] proved to be inaccurate.

Although the IPD-based covert channel is generally well-known [214, 10, 131, 137, 321], the problem of inaccuracies in timing enforcement during increased system load remained unsolved [133, 114]. We consider this complication in our design in Section 4.3.1 and present a compensation mechanism that detects and compensates unintended timing inaccuracies. Also, while most works simply assume that packets are of constant size [320] or padded to the maximum desired size [336, 131], our adoption of multiplexing and fragmentation enables flexible packet size enforcement. The combination of different mit-
igation techniques makes our implementation the first prototype for comprehensive covert channel mitigation.

Regarding performance trade-offs, Mode Security was proposed as a general approach to adapt to resource usage by switching between different operation modes [63]. A similar approach called Traffic Stereotyping was proposed for networks [131]. To our knowledge, there is only one system that uses Mode Security to optimize covert channel mitigation, which aims to provide sender anonymity based on dynamic re-routing and IPD enforcement [320, 321]. They assume a trusted network stack on each network endpoint and a periodic global negotiation to achieve an equalized traffic matrix [320]. A performance analysis was done based on statistics collected from a medium-sized network [319]; however, no actual performance measurements of their system have been provided and the problem of determining the optimal enforcement mode was left unsolved. Alternatively, NetCamo [137] requires its endpoints to explicitly request their delay and throughput demands beforehand. We extend on these works by proposing a practical algorithm to determine the optimal operation mode on-demand. As we do not require sender-anonymity, problems of mix-networks do not apply to our approach (e.g., [279, 22]).

In contrast to probabilistic traffic obfuscation schemes criticized in [98], our framework enforces an information-theoretic bound on the maximum information leakage. Covert channel detection schemes such as [53, 130] are complementary to our work and can be useful to increase the attack complexity for the adversary, especially in cases such as the PktSort and PktDrop characteristics, where mitigation is costly.

While we know of no practical performance measurements for comprehensive covert channel elimination, an overhead of 45%-56% was reported for obfuscating the packet size in website traffic alone [193, 330].

### 4.6 Summary

We have motivated the problem of covert channels in Virtual Private Networks (VPNs) and presented the design, implementation, and performance of a covert channel-resilient VPN. We identified several covert channels and presented new countermeasures. We have investigated the problem of on-demand adaption of operation modes and presented an implementation for comprehensive, efficient covert channel mitigation in the Linux IPsec stack. Our evaluation shows that on-demand rate adaption is feasible and practical even for highly unpredictable traffic. In more predictable throughput benchmarks, our system achieves remarkable 169 Mbit/s in a 201 Mbit/s VPN connection (84%).

However, not all usages allow the deployment of VPNs to encapsulate all communication. In many cases, secure channels are established spontaneously with random endpoints, and significant portions of the involved traffic such as DNS queries and intermediate websites may be transmitted in the clear. In the next chapter we study the tethering detection as one popular current instance of this problem. We investigate what kind of information powerful adversaries can infer without actively disrupting the connectivity of large user bases, and how susceptible the existing Internet protocols are to mitigations approaches such as normalization and obfuscation.
Chapter 5

Tethering Detection and Countermeasures

The success of smartphones is having a tremendous impact on the usage and development of mobile phone networks. On the customer side, low prices and the ability to run sophisticated applications result in a perpetual use of Internet services, such as email, video streaming and social networks. Ubiquitous Internet connectivity enables business professionals to access company resources while traveling, transforming idle time in airports, trains, and hotels into office hours. Since local WiFi connections in hotels or airports are often expensive and sometimes even unavailable, customers are tempted to “tether” their laptops and tablets to the mobile network connection provided by their smartphone. Already, many commodity smartphones such as Android phones or Apple iPhones have integrated mechanisms for sharing the network connection with other devices, or can be modified to do so [152, 105].

On the provider side, the rapid growth of data usage on devices like tablets and smartphones has incurred large investments to optimize and reduce traffic load on the network infrastructure. Systems and methods are being developed to dynamically optimize hosting locations of content, broadcast data in 3G networks and to adjust the bit rate of content streams based on type and network load. In this setting, unexpected network usage can induce significant network overhead, breaking network optimizations and cost calculations. As such, tethering imposes a significant burden on mobile communication networks, a cost factor that providers like to accommodate in appropriate data plans\(^1\).

Thus, tethering is often prohibited in private smartphone contracts, and providers recently started to offer data plans that explicitly allow tethering at extra cost. For example, the AT&T “unlimited” data plan does not include tethering. Several customers, suspected of tethering, were informed to be switched to a different data plan, unless they report back to AT&T committing to stop their tethering [149]. Similarly, Verizon recently started to detect and redirect tethering users to their website, asking them to switch to a tethering data plan [164]. However, this rather invasive enforcement of unexpected contractual limitations, by deploying tethering detection on the provider side, is often met with resistance and many users attempt to hide their tethering from the provider [77].

In the following, we study the impact of information leakage and side-channel attacks

\(^1\)Private communication with provider employees.
for the case of tethering detection. We consider the cost of such large-scale traffic analysis and investigate the feasibility of lightweight traffic obfuscation and normalization mechanisms that are practical on modern embedded devices. After introducing the general problem of tethering and tethering detection in Section 5.1, we classify possible detection methods in Section 5.2 and assign cost factors to them based on the respective associated effort or cost for the provider. In Section 5.3, we then present efficient mechanisms that can defeat the vast majority of practical tethering detection mechanisms. We verify the feasibility of our approach with an Android-based prototype implementation. The main result of this work is that targeted traffic normalization and obfuscation is very practical on modern power-constrained embedded systems, and significantly more efficient than a generic VPN encapsulation.

5.1 Problem Description and Model

We consider the tethering detection scenario as illustrated in Figure 5.1. The subscriber uses a mobile station (MS) to connect to the mobile broadband network of the provider. The MS is a highly customizable smartphone that may be used by the subscriber to connect additional tethered clients (TC), to share the mobile network connection. The provider aims to prevent such usage by classifying the network traffic of its subscribers (customer base) into either tethering or non-tethering traffic. The undesired tethering traffic can then be blocked or directly billed according to the provider's policy. On the other hand, the subscriber aims to hide its use of tethering, i.e., to confuse or circumvent the tethering detection of the provider.

Note that there are also cases of “tethering” where the subscriber shares the mobile connection of the provider not only between own devices but also with third parties. Alternatively, more powerful mobile devices may (soon) be running virtual machines, creating a “virtual” tethering system. From the perspective of the provider, it is hard to

\[\text{Figure 5.1: Tethering Detection Scenario}\]
distinguish these types of tethering from each other. However, since all of these approaches result in the same network setup and incur similar load on the mobile provider's network, we assume that they are equally undesirable.

The subscriber could also deploy a VPN tunnel to hide connection details from the provider. However, VPNs introduce configuration and compatibility issues, and increase network delay due to overlay routing. VPNs are also easy to detect, so that a widespread use of VPN to hide tethering may result in a ban of VPNs for cheap data plans.

5.1.1 Adversary Model

For the purpose of this investigation we treat the mobile provider as the adversary. We assume that the provider has full control over the network connection of the subscriber and may re-route, insert, modify or block transmitted packets. Additionally, the provider is able to read all transmitted data, including application layer information, except in cases where the subscriber uses end-to-end encryption mechanisms like SSL or VPN.

However, the provider is subject to certain practical limitations: The subscriber has full control over the programs running on the mobile station MS and any tethered client TC. Moreover, manipulation of application layer content or active OS fingerprinting may be detected by the subscriber and regarded as an attack, which we consider costly for the provider. Some attacks also require more resources than others, such as application layer inspection or stateful tracking of connections, increasing the cost for tethering detection.

We model these limitations by considering the cost or practicality of the attack ("effort") with regard to the various criteria that may be relevant for the mobile provider. In particular, we rate the effort as "low", "medium", "high" for the following criteria:

**Impact Type:** We distinguish passive, active and destructive detection methods. Passive methods simply monitor transferred network traffic, such as TCP/IP source and destination header fields, and incur low effort. In contrast, active attacks manipulate the data that the provider transfers on behalf of the subscriber or inject custom packets to prompt a reaction from the subscriber. This generally requires more resources due to realtime traffic processing and tracking, which we rate as medium effort. Some active attacks can result in noticeable interruption of ongoing network communication for the subscriber and thus are not suitable for large-scale scanning of a provider's customer base. We consider such destructive attacks impractical.

**Protocol Layer:** We categorize attacks as network layer (low), application layer (medium) and "behavior layer" (high). Application layer attacks are generally more costly than network layer attacks since more complex protocol parsing and interaction is required, while lower layer protocols can be processed by most common hardware. With "behavior layer" attacks we denote the collection of traffic meta-data, which can encompass simple characteristics like timing and size of packets as well as more complex connection patterns. We rate the effort for large-scale scanning as high.
since the collection of useful meta-data requires long-term observation of individual subscribers.

**Privacy Violation:** We distinguish attacks that are not privacy-critical (low) from those that work with privacy-sensitive data (medium) (e.g., inspection of application layer data) and attacks that modify or inject application data (high). The latter approaches are often problematic with regard to data protection laws, and especially the undesired modification of user data is strictly prohibited in many countries [163].

**Pre-condition:** We differentiate between unconditional (low) or conditional (medium) attacks. Unconditional attacks can always be launched, e.g., traffic analysis or querying the MS for active fingerprinting. In contrast, conditional attacks such as special HTTP or DNS access patterns depend on the behavior of the targeted subscriber and may be more or less likely to occur. Note that some pre-conditions may be easily met by longer-term passive observation, while others are only realistic if the provider can actively manipulate the MS, such as sending uncommon IP packets for OS fingerprinting. If a pre-condition is unlikely to occur even in active attack scenarios we denote the attack as impractical.

**Detection Effort:** We categorize detection methods based on signatures (low), heuristics (medium) or profiles (high). Heuristic and profile-based methods suffer from increased costs due to the computational efforts required for traffic processing, and profile-based methods are the most expensive as they additionally need traffic profiles to classify collected traffic with high accuracy. For example, network layer fingerprinting uses a simple database lookup (‘signature’) to classify OS implementations and is generally cheaper than traffic classification with statistical analysis and machine learning.

Using this classification, we derive the overall effort for each attack as the maximum of the efforts for each particular criterion. For example, the effort of evaluating the HTTP User Agent header is medium: It is a low-impact attack (low) but the inspected data is on application layer and may be considered privacy-sensitive (medium).

### 5.1.2 Communication Architecture

We assume a standard 2G/3G network setup on the provider side and a regular TCP/IP LAN\(^4\) on the subscriber side. The mobile station MS dials into the provider’s network and provides multiple interfaces such as wireless LAN, Bluetooth and USB to connect to local devices of the subscriber.

**Tethering Technology**

The tethering mechanism that connects the subscriber’s LAN to the provider’s WAN can be implemented in several ways. Most commonly, the MS is used either as a modem or

\(^4\)Note that we use the term TCP/IP in this work to refer to the complete TCP/IP protocol stack with UDP and ICMP.
IP gateway. Historically, the modem solution was used as it requires the least resources on the MS. However, the IP gateway solution is preferred on today’s smartphones as it allows the simultaneous use of voice and data services. Some applications also offer other tethering techniques like application layer proxies or port-forwarding. However, such solutions provide only limited connectivity and are not easily deployed in combination with, e.g., VPN or VoIP software. Hence, we focus on the case of tethering where the smartphone acts as an IP router and gateway for the LAN, forwarding IP packets between LAN and WAN.

Technically, IP gateways for tethering on mobile phones are implemented using Network Address Port Translation (NAPT) [285], also known as 1:n Network Address Translation (NAT). In 1:n NAT the port numbers and request identifiers of UDP, TCP, ICMP and other protocols are used to multiplex connections from the private LAN IP address space to the single, public IP that is typically issued to the mobile station MS by the provider. The deployment of NAT has three major consequences for tethering detection: (1) NAT transforms forwarded IP traffic, resulting in a modified traffic pattern that may be used to directly detect NAT; (2) a tethered client TC is not directly reachable from the WAN, so that, e.g., active fingerprinting by the provider will always only detect the MS itself but not TC; (3) since NAT is designed to be transparent, TCP and UDP payloads are transmitted unmodified, resulting in several options for tethering detection at application layer.

Mobile Networks Architecture

Figure 5.2 shows an overview of the 2G/3G mobile communication architecture. On the left, the MS is connected to one or more tethered clients via, e.g., wireless LAN. The mobile station acts as an IP router with NAT for these LAN clients, forwarding their IP packets through the provider’s 2G/3G network in a General Packet Radio Service (GPRS) tunnel. The provider’s Base Transceiver Stations (BTSs) forwards the GPRS frames on link layer to the Base Station Controller (BSC) and then to the Serving GPRS Support Node (SGSN) [19]. Once in the provider’s network, GPRS frames are separated from voice traffic and forwarded to the Gateway GPRS Support Node (GGSN), where they are finally forwarded to the Internet [20].
CHAPTER 5. TETHERING DETECTION AND COUNTERMEASURES

Figure 5.3: Classification of tethering detection mechanisms. Tetherway includes defenses against all low and medium effort detection techniques.

As can be seen in Figure 5.2, the lowest protocol layer that is transported all the way from the MS through the provider’s network is the topmost IP layer. Any lower layer information in the subscriber’s LAN/WLAN is discarded already at MS, where forwarded TCP/IP packets are encapsulated in GPRS frames in the same way as locally generated (“non-tethered”) packets, and are thus indistinguishable on link layer. Although the IP layer information can have some effect on lower layers, such as frame length and timing, these characteristics are also extractable at the IP layer. Hence, we reduce our analysis of tethering detection mechanisms to IP and higher layer protocols used by MS and TC.

5.2 Detecting Tethering

Superficially, tethering detection appears similar to well-known mechanisms from network or OS fingerprinting and analysis of application behavior. However, most common fingerprinting techniques are not actually applicable and more complex attacks like active manipulation or traffic analysis quickly become too costly to be applied to the huge customer base of a mobile provider.

A general overview of the different types of attacks is provided in Figure 5.3. Abstractly, we can classify the possible tethering detection mechanisms into (1) network layer fingerprinting, (2) application layer inspection and manipulation, and (3) behavior and traffic pattern analysis.

In the following, we discuss each of these approaches in detail and rate their feasibility and practicality by assigning costs based on the adversary model in Section 5.1.1.
5.2. DETECTING TETHERING

5.2.1 Network Layer Attacks

Tethering detection methods on network layer can be generally described as fingerprinting attacks. However, we must emphasize that tethering detection is different from the well-known OS fingerprinting attacks: In OS fingerprinting, the adversary aims to detect the OS type running on a specific remote machine, while in tethering detection we are interested in identifying additional hosts behind the MS, or the fact that Network Address Translation (NAT) is enabled at the MS. Hence, many standard attacks are not effective while other and new attacks become useful. In the following we thus distinguish passive attacks and active attacks that are based on (i) requests, (ii) injection, and (iii) manipulation.

Passive Fingerprinting

Passive observation of network and transport layer header fields and traffic flows can be used to distinguish hosts behind NAT or directly detect the use of NAT. In the IP header, the fields for Differentiated Services (DS), Explicit Congestion Notification (ECN), IP Flags and especially Time To Live (TTL) may be used differently depending on the OS that the packet originates from (e.g., [49, 288, 203]). Similarly, information on the Initial TCP Window Size and Sequence Number, the supported types, values and order of TCP Option fields such as Maximum Segment Size (MSS), Window Scaling and Timestamps can be used to discriminate different TCP/IP implementations behind NAT [55, 293, 66]. We consider such passive network layer attacks as low effort, as they are the least invasive and most scalable.

Fingerprinting by Request

Most established OS fingerprinting techniques assume an active adversary to query the target with specially crafted TCP or ICMP requests [282, 34]. However, such requests are usually answered by the mobile station MS itself, and thus can only identify the MS and not the TCs. To the best of our knowledge there are also no active fingerprinting attacks that directly detect the use of NAT by querying the MS. Hence, active fingerprinting attacks based on requests are generally ineffective, as long as they only target the MS itself and cannot detect its use of NAT, so that we denote such attacks as impractical.

In the following we discuss two different approaches for active fingerprinting which either inject or manipulate packets of existing connections. The NAT at the MS keeps track of such existing connections and will forward injected or manipulated packets as long as they can still be recognized as part of a known ongoing connection\(^5\). Note that this is an inherent part of the operation of the NAT gateway to detect and associate packets of active connections, and that it is not typically able to distinguish correctly injected packets from regular IP traffic.

---

\(^5\)In TCP and UDP, the connection is identified by the source and destination ports and IPs addresses. ICMP packets are recognized by their identifiers or, in case of ICMP error messages, based on the UDP/TCP port and IPs contained in their embedded TCP/UDP fragment [285, 286].
CHAPTER 5. TETHERING DETECTION AND COUNTERMEASURES

Figure 5.4: ICMP injection attack for detecting NAT.

Fingerprinting by Injection

An active fingerprinting attack can traverse the NAT barrier by injecting packets into an existing connection, so that they are recognized and translated by the NAT engine. For the UDP protocol, no such attacks are known in the related work, likely due to its inherent simplicity. TCP packet injection attacks are possible but problematic, as packet injections desynchronize the TCP connection between the original sender and receiver [341]. This requires the provider to constantly manipulate the TCP packet stream until an opportunity for re-synchronization occurs, or until the connection is ended, or otherwise results in a noticeable interruption of the TCP session of the subscriber. We rate the resulting longer-term realtime traffic manipulation as a high effort attack.

The remaining network layer protocol that is frequently used today is ICMP. For active fingerprinting of hosts behind NAT, an ICMP error message could be injected from the provider based on an existing UDP or TCP connection. A NAT engine is required to statefully rewrite and forward several types of such messages to ensure transparent IP operation [286]. However, ICMP errors often signal critical faults in the forwarding of IP packets and are thus not designed to generate an observable response. In fact, most ICMP errors will immediately terminate the respective UDP or TCP connection, leading to noticeable interruptions that make such approaches impractical.

We have identified only one error message that is (1) regularly forwarded into the LAN behind a NAT gateway, (2) handled by standard TCP/IP implementations, including Android, and (3) results in changes observable to the provider if handled by a tethering client: the ICMP “fragmentation needed and don’t fragment bit was set” error message.

As illustrated in Figure 5.4, the provider may inject such a message as a response to an ongoing TCP/IP connection. The error is forwarded by the NAT gateway, manipulating one of the tethered hosts to believe into a smaller Maximum Transmission Unit (MTU) for this particular IP route, i.e., for the IP layer connection between the source and destination of the respective TCP or UDP connection. As a result, the target of the injected ICMP error (TC or MS) will start transmitting smaller packets for that particular IP route, which can be observed by the provider. But since the ICMP error is regularly forwarded to only one of the potentially multiple hosts on subscriber side, the provider can then detect the use of tethering by checking if all other connections using the same IP route adopt the same reduced MTU.
Note that such ICMP errors may legitimately occur whenever the Path MTU (PMTU) is smaller than the currently used MTU. The MS cannot simply discard them as that would prevent the respective TC from learning about the bottleneck for that particular connection, resulting in unclear packet loss and irritating effects such as partially loading websites with subsequent network timeout. While the MS could enforce the smallest possible MTU from the very start and thus prevent any legitimate PMTU errors, that would significantly impact network performance and in itself indicate the use of tethering software to the provider. Hence, there seems to be no efficient solution for the MS hide the existence of multiple TC in this scenario.

A drawback of this detection method is that it only works if multiple IP connections to the same destination hosts are opened at the same time by different tethering hosts, as otherwise no difference in the behavior of any two connections can be observed. While this pre-condition cannot be easily induced by the provider, it is also not very unlikely to occur, especially when considering the high frequency at which email, instant messaging and news aggregation clients connect to popular Internet services today. We classify such ICMP injection as a medium effort attack due to its active manipulation of traffic and the required pre-condition.

Fingerprinting by Modification

The adversary may also manipulate IP and TCP headers that are destined for the MS and potential TCs with the goal to create observable changes in the connection state or behavior. The scenario is similar to the ICMP MTU attack illustrated in Figure 5.4: When manipulating the IP or TCP headers of an ongoing TCP/IP connection, the manipulation propagates through the NAT barrier at the MS. The tethered clients TC₁, TC₂,... may then act differently depending on the included options, creating a client-specific feedback that is observable by the provider.

One example of such an attack is to set the TTL for inbound packets towards the MS to "1". Such packets can be received by the MS, but an additional forwarding to the TCs would decrement the TTL to "0", so that the packet is dropped at the NAT gateway. Similarly, the Flags field or Header Checksum of the IP or TCP header may be manipulated to detect different types of operating systems through their different ways of error handling. However, all these manipulations involve a high risk to noticeably interrupt ongoing connections, making them impractical for scanning of large customer bases.

There are also several less common optional headers in IP and TCP that may be handled and supported differently by different TCP/IP implementations. The provider may exploit the difference in endpoint behavior to detect the existence of multiple hosts at the MS, by injecting TCP options into existing TCP/IP streams and observing the response behavior. Due to their optional nature, the injection or deletion of such options will usually not break the connection, however, they still require an active manipulation and stateful observation of the traffic flow, resulting in medium effort attack.
5.2.2 Application Layer Attacks

A large number of application layer characteristics can be used to differentiate hosts behind a NAT, either explicitly based on meta-data information in protocol headers or implicitly, by exploiting the different features supported by individual applications. However, the line between mobile and desktop “Apps” becomes increasingly blurred by the rapid progress in smartphones and particularly tablets, which are getting as complex and powerful as desktop systems.

Application Data Analysis

The easiest way to identify the number and type of hosts and applications on application layer is passive application layer inspection or Deep Packet Inspection (DPI). Well known examples for application layer data that identify systems and applications behind the MS are the User Agent field in the HTTP header and the host identifier strings sent as part of eMail or Instant Messaging (IM) communication [56]. Protocols with more complex negotiation of features and algorithms, such as TLS or IPsec, also often exchange an implementation-characteristic sets of supported features during their negotiation phase, which potentially allow distinguishing multiple hosts behind a NAT gateway even if the actual user data is encrypted.

Passive application layer attacks are highly practical. The required DPI can be done asynchronously and is already available for several policy-enforcement scenarios in the network management. Hence we can rate this approach as medium effort.

Application Layer Injection

For active attacks, i.e., the injection of code into transferred websites or the redirection of users to the provider’s servers, the attack surface for distinguishing TCs is practically unlimited. However, such active attacks are also rather resource intensive, have a high risk of getting noticed and may be interpreted by the client as intrusion and privacy invasion [164, 246]. According to our cost model, the effort for such attacks is therefore considered to be high due to potential privacy violations.

5.2.3 Traffic Metadata Analysis

The third main category of tethering detection concerns the analysis of traffic metadata. Similar to application layer attacks, the area of statistical traffic analysis is very large [71]. However, we can single out two approaches that result in rather efficient detection mechanisms, while the more sophisticated traffic analysis with machine learning is more resource intensive.

We emphasize that the popular packet size and timing analysis usually requires long-term observation and sophisticated machine learning techniques to classify traffic with reasonable accuracy. As such it constitutes a rather high effort attack when applied to the large customer base of a mobile phone provider.

---

See, e.g., http://browserspy.dk and the demonstration at https://panopticlick.eff.org/
5.2. DETECTING TETHERING

**Endpoint-Specific Services**

Many operating systems and applications can be identified based on the individual Internet services they use. For example, Android phones are unlikely to connect to the Microsoft Windows update servers or to package repositories provided by most large Linux distributions. Instead, they will mainly connect to the Android Market for software update information, and to the configured Google, Exchange or Facebook accounts for synchronization of contact data, etc. Similar considerations apply to individual applications such as anti-virus scanners, office suites, PDF viewers and Java runtime environments that are known to regularly contact the servers of their respective vendors.

A special case of this category are tethering applications provided by the mobile provider itself, which explicitly signal the use of tethering by switching the provider’s Access Point Name (APN). These applications are regularly shipped with iPhones, but also with many Android phones.

Endpoint-specific tethering detection requires only passive traffic monitoring on network layer, without any particular post-processing of data. While the considered events (pre-conditions) are usually not easily to trigger by the provider, they are still likely to occur. For example, many systems check for software updates as soon as an Internet connection is established. Hence we rate this kind of attacks as low effort.

**Statistical Analysis**

With this category we denote approaches that use simple frequency or random distribution analysis, i.e., which employ rather simple analysis models [54, 85]. Well-known examples of this category concern the random distribution or range of values in individual network layer header fields, such as the IP ID fields [288, 49] or the clock skew in values of the TCP timestamp header [66]. We also found that the port-multiplexing of NAT leads to distinctive changes in the source port range and distribution. In contrast to previous works [288], we found this characteristic to be highly suitable for tethering detection (see Section 5.3.2 and 5.3.3).

A common property of these attacks is that they concern only lower-layer protocols and require only short to medium observation time. Thus we consider their overall cost as medium.

**Machine Learning**

Apart from the direct evaluation of header fields and connection states, a major approach in the classification of traffic is the traffic pattern analysis with machine learning. This approach is characterized by collection and model-based analysis of network meta-data such as traffic volume, packet size and timing, as well as more complex patterns like the number of simultaneous TCP connections for given destinations, or with specific higher-layer protocols.

An example attack that could be mounted in this way to reveal tethering is the detection of multiple browser caches at the client, which results in different HTTP object request patterns. Alternatively, the provider may attempt to distinguish particular TCP
flow control algorithms implemented at the MS, based on how individual TCP connections increase or throttle their packet rates over time.

The data processing phase involves the use of generalized profiles that are created by observing and then either implicitly or explicitly classifying large amounts of data (supervised or unsupervised learning, see [222, 332]). However, supervised learning involves a high effort in the learning phase, and the results of unsupervised learning require similarly high effort to confirm potential matches and filter false positives [175]. In the end, all types of learning-based traffic analysis appear to require long-term observation, careful system analysis and post-processing. Due to the high costs in regard to Detection Effort and Protocol Layer we rate their overall effort as high.

### 5.3 Defeating Tethering Detection

To evaluate the practicality of tethering detection, we have developed a generic architecture for normalization and obfuscation of the tethering characteristics identified in Section 5.2. In this context, normalization describes the process of modifying the distribution of values for the previously identified characteristics, such that they are close or identical to the distribution of a non-tethering system. We have implemented a prototype to defeat against the most practical (i.e., cheapest) attacks identified above and compare its performance against simple VPN solutions.

#### 5.3.1 Tethering Normalization Architecture

Figure 5.5 depicts our approach to normalization and obfuscation of tethering characteristics in network traffic. There are two main components for traffic normalization: (1) the Application Layer Filter and Cache and (2) the TCP/IP Normalizer with Masked NAT. A packet filter is used to assign data streams to the respective components, and after processing all packets are forwarded through the same WAN interface to the provider, as discussed in Section 5.1.2.

The Application Layer Filters must be developed specifically for each individual application layer protocol, like HTTP, and can thus also normalize application-specific patterns such as the User Agent meta-information in HTTP headers. Moreover, they can block, aggregate and cache queries to obfuscate access behavior and traffic patterns or imitate desired patterns. After the application layer payloads are processed, they are forwarded...
using the local IP handler of the MS, so that no TCP/IP layer normalization is required. The main disadvantage of this approach is that it must be implemented for each individual application protocol and cannot handle unknown protocols.

Hence, we have added the TCP/IP Normalizer as a generic network layer normalizer in case no Application Layer Filter was defined for a particular data stream. This also includes many protocols where application-layer processing is not possible or not deemed necessary, such as encrypted traffic. As a network layer normalizer, its capabilities are limited to the normalization of header fields and filtering of unusual requests and header options, as detailed in Section 5.3.2. The normalization is followed by the Masked NAT component, which translates the IP address range of the LAN traffic to that of the WAN traffic [285]. The NAT is masked in the sense that the range and distribution of the modified header fields are indistinguishable from that of the local IP stack (cf. Section 5.3.2).

### 5.3.2 Tethering with Tetherway

To verify the practicality of network normalization on smartphones, we implemented an Android App for tethering normalization system called Tetherway. Tetherway is based on the popular Android App android-wifi-tether\(^7\), which can use the ad-hoc wireless network, USB or Bluetooth personal network of the MS to connect to tethered clients, providing a standard IP gateway with DHCP and NAT. In the following we describe the implementation details of the TCP/IP Normalizer, Masked NAT, and two Application Layer Filters for DNS and HTTP. We denote packets that are destined to the Internet or provider network as outbound packets, and packets destined for the MS or TCs as inbound packets.

#### TCP/IP Normalization

We have implemented a TCP/IP packet normalization using the libnetfilter-queue extension\(^8\) of the Linux firewall subsystem. The extension allows us to program custom packet filters in the Linux userspace, enabling arbitrary rewrite of network packets to simulate non-tethering behavior.

**IP Header Normalization.** On IP layer we reset the DS field for outbound packets and enabled ECN. We set the TTL field to 64 and also reset the IP flags and fragment offset to disable fragmentation, as this is the default behavior in Android smartphones. Note that while NAT cannot handle fragments, Linux transparently defragments packets before NAT processing so that they are treated in the same way, regardless of whether they are destined for the MS or TCs.

More involved is the adjustment of the IP ID field to elicit the same random distribution as the IP ID values of a standard Android platform without NAT, i.e., a randomized initial value that is incremented with each packet of the same TCP/IP session. To imitate this behavior of modern Linux kernels we have implemented a corresponding stateful rewrite of IP IDs for TCP/IP sessions, using local copies of the respective randomization functions in the Linux kernel. For UDP, no normalization is required since the standard

\(^7\)http://code.google.com/p/android-wifi-tether/
\(^8\)http://netfilter.org/projects/libnetfilter_queue/
behavior to set the IP ID to zero is the same as it is done by regular NAT in Linux. Only in case of DNS requests, the IP ID may contain a random number to mitigate DNS cache poisoning attacks [177]. However, we have deployed a caching DNS proxy for this case. Finally, our system filters all inbound and outbound IP options as they are usually not used or needed.

**ICMP MTU Exceeded Injection.** The ICMP injection attack we proposed in Section 5.2.1 can be partly mitigated by replicating ICMP errors on the NAT gateway: Similar to the approach of IPsec, the gateway can record received ICMP MTU errors from the WAN and distribute them not only to the respective LAN host referenced in the error message, but also to all other hosts that send a packet larger than the reported MTU to the same destination IP. As a result, the provider will not receive any packets larger than the MTU size previously injected. The provider could still wait for the reduced MTU values to time out on the individual TCs, an event that occurs at different times depending on the deployed OS at the TCs. However, this event occurs only after 1-2 minutes, complicating detection as many TCP connections are not sufficiently long-lived.

Hence, while our defense is not perfect, the attack cost is increased by requiring long-term observation and the pre-condition of multiple long-lived TCP connections is less likely to occur, so we rate the new effort for this attack as high.

**TCP and UDP Header Normalization.** Figure 5.6(a) and 5.6(b) give an overview of the normalized fields in the typical TCP/IP headers. For the TCP header, we normalize the Sequence Number field and the ECN and Reserved Flags, as well as several TCP option headers. The Acknowledgment Number, Window Size and Checksum fields are also updated as a consequence of other corrections.

Just like in the IP header, the Explicit Congestion Notification (ECN) flags can be safely reset to always enable ECN support: ECN is supported by all modern hosts, and systems that do not support ECN suffer the resulting performance penalty in any case.

To normalize the TCP Initial Sequence Number (ISN), we statefully track TCP connections and use the functions for randomized ISN selection from the Linux kernel to select a new ISN for each TCP connection over NAT. Since the endpoints of a TCP connection rely on the sequence numbering for ordering and acknowledging received TCP
segments, we then record the offset between the original and newly chosen ISN for each new connection and adjust (1) all subsequent Sequence Numbers on outbound and (2) all Acknowledgment Numbers on inbound packets accordingly. Hence the provider is unable to distinguish ISNs for tethering, and all Sequence Numbers are correctly rewritten regardless of the packet reordering.

**TCP Option Headers.** We must normalize the various TCP Option headers to mitigate the problem of fingerprinting by modification (cf. Section 5.2.1). For this purpose, we purge all TCP Options that are not used by Android from inbound as well as outbound packets, i.e., all Option headers except for the Message Segment Size (MSS), Selective ACK (SACK), Timestamp (TS), Window Scale (Wscale) and No Operation (NOP) fields. For outbound packets, we furthermore normalize the order and content of the remaining supported TCP Options.

The MSS Option can be normalized using the `iptables clamp-mss-to-pmtu` option. The SACK and Wscale Options are used only in the initial TCP handshake to signal support for selective acknowledgments and window scaling. The SACK Option does not contain any actual values and thus does not require normalization, except for its location within the TCP header. However, the Wscale Option on mobile stations often advertises the smaller Window Scale factor than the one used for the LAN and WLAN interfaces of desktop systems. To transparently rewrite this value, we statefully track the TCP connection, recording the original Window Scale factor before resetting it to the typical Wscale factor of “1” for Android. For all outbound TCP headers, we recompute (left shift) the Window Size header value to compensate for the smaller Window Scale factor. In the worst case, this modification reduces the absolute receive window assumed by the sender, possibly reducing TCP performance. However, this is also the expected behavior for a regular (non-tethering) MS. Finally, the TS Option adds two timestamps to all TCP packets to let endpoints compute the precise Round-Trip Time (RTT) of a connection [157, 66]. To make the TS values indistinguishable from the timestamps of the MS, we simply replace all timestamps inserted by the TCs with values generated by the MS. The resulting change in the computed RTT is negligible, as TC and MS are typically very close.

Apart from the SACK Option, all the discussed Options could be synthesized on the gateway without breaking the TCP session. However, currently we treat a missing TCP Option as an error and do not currently implement such synthesis.

**Masked NAT.** The source port multiplexing by NAT, or more specifically Network Address Port Translation (NAPT), can lead to characteristic source port distributions: Firstly, the standard Linux NAPT implementation uses a different port range than the standard range for ephemeral source ports in outbound connections. Moreover, Linux NAT tries to preserve the original source ports whenever possible, so that source port distributions of different TCs are likely to remain visible regardless of NAT (cf. Figure 5.7).

Fortunately, the Linux NAPT subsystem also supports two parameters to set the port range and enable port randomization. By source code inspection we confirmed that enabling port randomization disables the preservation of source ports, and that the employed randomization function is the same as the one used in the ephemeral source port selection for local traffic. Hence, we can eliminate differences in the distribution of
CHAPTER 5. TETHERING DETECTION AND COUNTERMEASURES

Figure 5.7: Source port distributions for Android and Windows 7 without NAT, with NAT and with Tetherway. An obvious difference between Windows and Android can be seen. In contrast, the shown port distribution of Windows 7 behind Tetherway NAT is normalized to behave like regular Android systems without NAT.

source port values sent via NAT versus that of local connections by the MS, by enabling randomized port mapping and setting the same range of ports as used by the regular ephemeral port selection of Android.

Application Layer Proxies

We have implemented application layer filtering for the two most used services, DNS and HTTP. In both cases we have deployed standard application layer proxies to implement the filtering and caching required for eluding tethering detection.

For HTTP, we deployed Privoxy\(^9\) as a filtering proxy. Privoxy is used to anonymize HTTP traffic for users of Tor\(^10\) and provides extensive and well-tested rules for filtering, e.g., user tracking based on cookies and Webbugs, or embedded active content such as Adobe Flash. To obfuscate object request patterns on the behavior layer, the filtering proxy is backed by a caching parent proxy, Polipo\(^11\). As a result of this construction, the HTTP request behavior becomes similar to that of an endpoint with only a single browser cache, as it would be expected from a non-tethering MS.

For DNS, we use dnsmasq\(^12\) as a local DNS cache on the MS. We implement a simple DNS filter by returning the local host IP address 127.0.0.1 for blacklisted DNS records. This can be used to block behavioral patterns like accesses to Windows Update or the standard Windows time synchronization servers.
5.3. DEFEATING TETHERING DETECTION

![Graph showing power consumption for Tetherway compared to regular tethering and VPN.]

Figure 5.8: Power consumption for Tetherway compared to regular tethering and VPN.

5.3.3 Evaluation

We evaluate the efficiency of our normalization engine by comparing the phone’s power consumption while downloading 960 objects (50MB total) from 2 websites using (1) regular non-normalizing tethering, (2) a typical VPN software and (3) our Tetherway prototype. In particular, we compared Tetherway against the android-wifi-tether App that it is based on, and against Juniper Junos Pulse VPN for Android. As can be seen in Figure 5.8, the total energy consumption and the amount of time of Tetherway for completing the same task are close to that of regular tethering and notably below that of a VPN client. This confirms our subjective impression that responsiveness of the VPN connection was notably lower, likely due to the additional hops introduced by the VPN tunnel.

To confirm the proper normalization of header fields, we compare the distributions of critical header fields such as the source ports in Figure 5.7 and TCP initial sequence numbers in Figure 5.9. In each case, a clear difference in the distributions of standard Android and a Windows 7 TC can be identified. Specifically, Figure 5.7 shows a highly predictable port usage for Windows 7 hosts, while Android uses a different port range with randomized source port selection. Similarly, the random distribution of the TCP ISN values shown in Figure 5.9 is distinctively different for Windows 7 and Linux, with Windows hosts using a larger range of values that are not changed by regular NAT. In contrast, the distribution of the Windows 7 client behind Tetherway is similar to the one expected from a non-tethering Android MS.

We conclude that our normalization is efficient and effective, making tethering detection much harder for the provider. VPN software can also be used to hide all identified tethering characteristics, by setting up the MS as a VPN gateway for the TCs. However,

---

9http://www.privoxy.org/
10http://www.torproject.org/
11http://www.pps.jussieu.fr/~jch/software/polipo/
12http://thekelleys.org.uk/dnsmasq/doc.html
5.3.4 Limitations

Tetherway focuses on the most common and most easily detected tethering characteristics, as illustrated in Figure 5.3. Several possibilities remain to identify tethering setups. Our HTTP and DNS caches do not enforce a particular packet timing or obfuscate patterns in the network access behavior of email or IM clients. However, such normalization can be added, e.g., using the Linux network emulator netem\textsuperscript{13}.

Furthermore, although HTTP and DNS traffic accounts for a large amount of device mobile traffic, many other protocols such as Simple Mail Transfer Protocol (SMTP), Internet Message Access Protocol (IMAP) or the several Instant Messaging protocols are also in widespread use. However, most modern applications employ encryption based on TLS, hiding any application layer characteristics.

Note that several more complex detection mechanisms, e.g., based on traffic analysis, may still be viable if the provider applies some preliminary filtering to reduce the number of suspects, e.g., based on the amount of traffic usage. However, an in-depth analysis of these approaches is outside the scope of this work.

5.4 Related Work

To our knowledge, this work is the first to consider the problem of tethering detection, which can be seen as a generalization of the previously considered NAT detection. We

\textsuperscript{13}See, e.g., tcn.hypert.net/tcmanual.pdf
systematize tethering detection methods and show that the most cost-effective techniques can be mitigated efficiently.

Vendors like Cisco and Sandvine already provide tethering detection solutions [82, 264]. However, they currently inspect only a rather simple mix of network and application layer headers, indicating that injection or traffic analysis attacks are indeed more costly. Consequently, first tethering Apps also provide correspondingly simple detection countermeasures like resetting the IP TTL field\textsuperscript{14} or using proxies\textsuperscript{15}.

Regarding the general problem of tethering detection, we identify three major categories of related work: (1) Revealing multiple hosts or different operating systems behind an IP gateway, (2) detecting use of NAT and (3) performing general statistical analysis of network traffic.

**Direct NAT detection.** The use of NAT and the counting of hosts behind NAT was previously considered as a general problem of network mapping and measurement. For example, it was proposed to detect NAT based on analysis of the IP ID field in the IP header and to count "NATed hosts" through reconstruction of IP ID sequences [49]. In another approach, a naive Bayesian classifier was used to detect hosts behind NAT based on IP TTL, DF, initial TCP window size and TCP SYN packet size [55]. Similarly, the authors of [288] propose to combine multiple parameters to increase detection accuracy, using data such as IP ID, TTL and source port distribution.

We consider these attacks among the most practical since they involve only simple statistical analysis on network layer data. We extend on these works by showing that source port distribution is a highly practical tool for detecting NAT, and that countermeasures for all previously presented network layer NAT detection techniques are feasible even on resource-limited devices (cf. Section 5.3.2).

**Fingerprinting and Scrubbing.** OS fingerprinting is a general technique to identify remote systems by the difference in their TCP/IP stack implementations [29, 117, 200], and corresponding software products are widely available.

However, as shown in our analysis, the most common active fingerprinting attacks, which work by sending specially crafted requests to the target, are ineffective in case of tethering detection (cf. Section 5.2.1). Similarly, most fingerprinting attacks that rely on the modification of header fields often break the affected connection, making them unsuitable for tethering detection by providers.

Multiple works propose normalization or scrubbing of network traffic, such as IP Personality\textsuperscript{16}, Morph [326] or ipMorph [243]. Network-based normalization has also been proposed to normalize traffic for processing in network intrusion detection systems [204, 140]. The main difference between Tetherway and existing solutions is the purpose of normalization, which determines the kind of distributions to imitate and in our case requires, e.g., the use of masked NAT, while defense against active fingerprinting by request is not needed.

**Traffic Analysis.** Traffic analysis has been used to detect network applications [292], online activities [338], behavior profiles of client systems [332] and properties of the encrypted network, namely routing and flows [85]. Furthermore, it was shown that traf-
fic analysis could yield results even if the transferred packets are encrypted or timing is masked [85, 331, 197, 54]. On the other hand, various countermeasures against traffic analysis have been proposed, ranging from network layer meta-data normalization [320, 330, 221] to efficient application layer obfuscation [199].

We believe that complex traffic analysis attacks are not practical for tethering detection with large customer bases. Instead, we only normalize header fields such as the TCP/IP source ports, which elicit highly characteristic patterns in case of tethering. While we deploy application layer proxies for simple obfuscation of packet timing and connection patterns, we defer a detailed analysis of possible novel attacks and countermeasures for later work.

5.5 Summary

We have presented the first general analysis and classification of tethering detection techniques. Our analysis indicates that tethering detection is a heuristic, highly fragile process, and many techniques are easily defeated by a modified mobile station.

VPN software can be used as a readily available tool to evade tethering detection, but reduces network performance notably and is easily detected by the mobile provider. In contrast, our Tetherway prototype closely imitates regular non-tethering traffic with modest overhead.

While the large amount of potential attacks makes comprehensive precautionary measures impractical, any particular detection mechanism can be circumvented using appropriate normalization or obfuscation mechanisms. As a result, we expect that any widespread use of tethering detection technology will quickly be countered through community effort, increasing the development cost at the provider until tethering detection itself is not cost-effective anymore.
Part III

Techniques for Trust Establishment
Chapter 6
Towards Provable Software-based Attestation

In recent years, sophisticated security extensions have been introduced in common computing platforms to increase their security assurance. The Trusted Platform Module (TPM) security chip is available for many PCs, laptops and even some tablet PCs targeting business users. Modern CPU models boast complex mechanisms for trusted execution environments such as Intel Trusted Execution Technology (TXT) or ARM Trustzone.

Unfortunately, while the criticality and sensitivity of embedded systems is massively increasing, currently deployed components for trusted computing are often too complex and expensive to be considered in the various low-cost, resource-constrained embedded systems. When deploying sensor networks for auditing and controlling of complex, security-sensitive systems such as cars, industrial plants and critical infrastructures, the integration of TPMs and trusted execution functionality is far beyond the system’s capabilities. Indeed, even well-established CPU features such as caching, virtualization and virtual memory are often not deployed in low-end systems to reduce hardware production cost and power consumption. In these environments, it is hard to provide generic, strong security mechanisms that resist application and operating system failures.

In this context, software-based attestation has become an appealing research topic for alternative, low-cost trust establishment mechanisms that allows a third party verifier to validate the integrity of the software configuration of another prover system. In contrast to traditional approaches of integrating trusted co-processors [235], the software-based attestation paradigm exploits the computational constraints of a platform’s hardware to provide a reliable report about its software integrity. For this purpose, software attestation deploys a randomized checksum code which temporarily uses the complete computational resources of the prover’s platform, assuring that nothing else can be executed within that particular time frame. The verifier can then validate the platform state of a device by (1) sending a randomization challenge, (2) measuring the time that it requires the prover to respond with the computed checksum, and (3) simulating the prover’s computations using the expected software state, to see if the required computation time and result match that of an honest prover.

However, a major problem of current approaches is that no formal modeling and security proofs of the security properties of the underlying core functionalities and the
overall schemes. This is particularly problematic as software-based attestation makes some rather unusual assumptions, and current implementations often combine algorithms with unclear and possibly insufficient security properties. Most significantly, the cryptographic requirements on the address generator and the checksum function that underly most existing software attestation schemes have never been formally investigated, and multiple attacks have been documented [275, 278, 74].

In this chapter, we present the first formal security analysis and systematic construction of software attestation algorithms. We first provide some background information on the concept of software information in Section 6.1. We then present a formal system model of software attestation in Section 6.2, followed by a generic software attestation scheme and its underlying assumptions in Section 6.3. We prove the security of our generic scheme under the given assumptions in Section 6.4. We then show a possible instantiation of software attestation in Section 6.5 and discuss how it fulfills the various required assumptions. We discuss and compare our results in Section 6.6 and Section 6.7, and summarize our results and open problems in Section 6.8.

6.1 Background on Software-based Attestation

The idea of software-based attestation can be traced back to the problem of tamper-proof remote execution of mobile agents [146, 147]. Such agents are deployed with a signed expiration date that can be “recharged” via code obfuscation on a trusted platform. The idea was extended in [284, 171] to let the executed code securely report the software state and to assure that this state is running on actual hardware, i.e., is not simulated.

Unfortunately, the initial approach of using the CPU state and caching effects, which are supposedly platform-specific and expensive to simulate, was shown to be vulnerable to several attacks [275, 278]: Redundancy in the code and simple hardware manipulations can be used to forge a valid attestation checksum within the required time. Furthermore, the proposed side-effects of the software execution are not reliably deterministic and often not available for inspection by software. Finally, [278] shows that the scheme of [171] is generally unsuitable for prover authentication, as it does not allow identification of the remote device or its user but only of a class of prover platforms.

However, the basic idea of using timed self-checksumming code to ensure correct execution, and to use that checksum at the same time to make statements about the state of the underlying platform, was shown to be suitable to provide a limited form of purely software-based attestation [275]. In contrast to previously known attestation mechanisms this approach does not require any explicit hardware support and thus appears ideally suited for attestation of low-cost embedded systems or legacy systems. As a result, several subsequent works investigated the suitability and extension of software attestation for a variety of computing platforms, including sensors, peripherals and voting machines [271, 192, 113, 273, 185].
6.2 Software Attestation Model

In the following we provide a generic model for software attestation. After introducing the notation, we propose a formalized system model and security goal.

6.2.1 Notation

For a finite set \( X \), \(|X|\) denotes the number of elements in \( X \) and for a bitstring \( s \), \(|s|\) denotes the bit length of \( s \). Let \( A \) be an arbitrary algorithm. Then \( y \leftarrow A(x) \) means that on input \( x \), \( A \) assigns its output to variable \( y \). Let \( B \) and \( C \) be two parties that can run interactive and probabilistic algorithms and that run a protocol \( \text{Prot} \). We denote with \((B : y_B; C : y_C) \leftarrow \text{Prot}(B : x_B; C : x_C; \ast : S)\) a protocol-run, where \( B \) and \( C \) are given a common input \( S \) and some input (denoted with the \( \ast \) symbol) \( x_B \) (resp. \( x_C \)) and generate an output \( y_B \) (resp. \( y_C \)). While running the protocol, \( B \) and \( C \) have access to the messages sent by the other party. Let \( E, F, G \) be some events, then \( \Pr[E|F \land G]\) denotes the probability that \( E \) occurs under the condition that event \( F \) happened before event \( G \).\(^1\) Let \( \mathcal{D} \) be a probability distribution over the set \( X \), then the term \( x \overset{\mathcal{D}}{\leftarrow} X \) means the event of assigning of an element of \( X \) to variable \( x \), which has been chosen according to \( \mathcal{D} \). Further, we define \( \mathcal{D}(x) := \Pr[x|x \overset{\mathcal{D}}{\leftarrow} X] \) for each \( x \in X \) and denote with \( \mathcal{U} \) the uniform distribution.

6.2.2 System Model

The focus of software attestation are resource-constrained embedded systems without security hardware (such as sensors). Hence, we are concerned with provers \( P \) that are embedded devices and we aim at designing a formal system model that is clear, intuitive and reasonable in practice.

A typical system architecture of \( P \) consists of memory and a Computing Engine (CE), as depicted in Figure 6.1.

The memory consists of Primary Memory (PM), such as CPU registers and cache, and Secondary Memory (SM) such as RAM and Flash memory. Each of these memories can store a limited number of memory words that are indexed with memory addresses \( b_j \) and \( a_i \), respectively. CE comprises an arithmetic and logic unit that can perform computations on data in PM and alter the program flow. For performance reasons, PM is typically fast but also expensive. Hence, the number of memory words that can be stored in PM is usually much smaller than the number of memory words \( s \) that can be stored in SM. To make use of SM, CE includes the \texttt{Read} instruction to transfer data from SM to PM and the \texttt{Write} instruction to write data from PM to SM. More precisely, \texttt{Read}(\( S, a_i, b_j \)) takes as input a memory address \( a_i \) of SM and a memory address \( b_j \) of PM and copies the data word \( x \) stored at address \( a_i \) in SM to the data word at address \( b_j \) in PM. For convenience, we write \texttt{Read}(\( S, a_i \)) instead of \texttt{Read}(\( S, a_i, b_j \)) whenever the

\(^1\)This is the common approach in cryptography to define probabilities and slightly different from classical mathematics.
address $b_j$ of PM is not relevant. Note that $\text{Read}(S, a_i, b_j)$ overwrites the content $d_j$ of PM at address $b_j$. Hence, in case $d_j$ should not be lost, it must first be copied to SM using $\text{Write}$ or copied to another location in PM before $\text{Read}$ is performed. It is important to stress that, whenever CE should perform some computation on some value $x$ stored in SM, it is mandatory that $x$ is copied to PM before CE can perform the computation. Furthermore, since SM is typically much slower than PM, $\text{Read}$ and $\text{Write}$ incur a certain time overhead and delay computations on $x$. Moreover, the $\text{Read}$ and $\text{Write}$ commands are slower (e.g., take more time/clock cycles) than other instructions. The program code that controls CE is encoded as part of the software state $S$, which is stored in non-volatile SM (e.g., Flash memory). Note that, in this work, we only consider provers as described above. The verifier $\mathcal{V}$ is an arbitrary computing platform.

**Remark 1: Memory Architecture.** The memory architecture of commodity computing platforms is usually more complex than illustrated in Figure 6.1. In particular, the available memory is typically organized in multiple layers, where the memory closer to the CPU is faster but, due to cost constraints, also significantly smaller than the memory farther away from the CPU. Faster memory layers are used to cache access to slower memory layers and many platforms feature additional hardware logic to predict and fetch memory blocks in advance. However, when accessing memory pseudo-randomly, the caching logic cannot predict and anticipate the memory accesses anymore. In these cases, any employed caching becomes effectively disabled [94], allowing us to exclude caching logic from our considerations without additional assumptions on processor complexity and capabilities.

**Remark 2: CPU Architecture.** Many CPUs feature instructions that can perform computations directly on data stored in the Secondary Memory (SM). However, due to the underlying physical construction of the CPU and memory hierarchy and the prediction accuracy of the pre-fetching logic used, such instructions still incur the overhead of
6.2 SOFTWARE ATTERTATION MODEL

A regular memory access, often several clock cycles more than a computation using data in PM [94]. Hence, our computational model in Figure 6.1 still applies. Note also that our system model focuses on low-end embedded systems and excludes high-end computing platforms with multiple CPUs and Direct Memory Access (DMA). These high-end systems are often already equipped with dedicated security hardware and in our opinion not an attractive target for software attestation protocols.

6.2.3 Secure Software Attestation

An attestation protocol must assure two fundamental properties about the prover. The verifier must gain assurance on which program code is stored on the prover and that a particular part of this code has been executed in the expected manner during the attestation protocol-run. The former is efficiently assured by computing a checksum over the desired program code (state), which we define as follows:

**Definition 1: State.** Let $l_s$ be the number of bits that can be stored in a memory word (state entry) of the Secondary Memory (SM) of the prover $P$ and let $s := 2^{l_s}$ be the number of memory words that can be stored in SM. We denote the state of $P$ with $S := (s_1, \ldots, s_k) \in \{0, 1\}^{l_s \cdot s}$, being the memory words stored in the SM. We denote the state of a prover $P$ with $\text{State}(P)$.

**Definition 2: Good State.** The state $S$ of a prover $P$ (Definition 1) is a good state for the verifier $V$ if $V$ knows $S$. All other states are denoted as bad states.

Ensuring that a certain program is currently executed by the prover is harder to capture. In particular, run-time attacks such as return-oriented programming attacks [251] illustrate that one cannot deduce which code the prover is executing solely based on what program code segments it stores. In (remote) attestation schemes, a trusted hardware component such as the TPM [311] or TXT [154], is used to ensure that a particular program code was started [235]. In contrast, software attestation exploits timing information to assure that a particular code is executed in a particular time frame $\delta$. We formalize the security goal of software attestation as follows:

**Definition 3: Secure Software Attestation.** An attestation protocol $\text{Attest}$ is $\varepsilon$-secure with respect to a set of good states $\Sigma \subset \{0, 1\}^{l_s \cdot s}$ (Definition 2) and a positive $\varepsilon \leq 1$ if $V$ is a probabilistic algorithm with the following properties:

Completeness: Let $\text{Time}_P(\text{Attest})$ be the time needed by $P$ to run $\text{Attest}$. $V$ accepts if the state of $P$ is a good state and $P$ finished $\text{Attest}$ within a certain time threshold $\delta$, i.e., for all $S \in \Sigma$ and all provers $P$ according to the system model (Section 6.2.2)

$$\Pr[\text{out}_V = \text{ACCEPT}|\text{State}(P) = S \wedge \text{Time}_P(\text{Attest}) \leq \delta$$

$$\wedge (P : -; V : \text{out}_V) \leftarrow \text{Attest}(P : -; V : -; * : S)] = 1.$$

Soundness: $V$ accepts only with a certain probability $\varepsilon$ if the state of $P$ is not a good state.
and/or $P$ exceeded time threshold $\delta$, i.e., for all $S \in \Sigma$ and all provers $P$

$$\Pr[\text{out}_V = \text{accept} | (\text{State}(P) \neq S \lor \text{Time}_P(\text{Attest}) > \delta) \land (P : -; V : \text{out}_V) \leftarrow \text{Attest}(P : -; V : -; * : S)] \leq \epsilon.$$  

Remark 3: Traditional Notion of Attestation. Two additional properties are usually associated with attestation. Attestation protocols typically authenticate the prover, enabling the integrity verification of remote systems (remote attestation). However, software-based attestation by itself is unable to store cryptographic secrets for authentication, and is thus mainly proposed for "local" attestation scenarios such as computer peripherals [192] or human-verifiable attestation of voting machines [113]. The second property is that attestation is often considered to imply what is being executed by the prover right after the attestation protocol-run, since knowledge about the past state of a deterministic machine implies knowledge of the current and future states. However, since most platforms are also subject to unknown, potentially malicious inputs, this relation is not straightforward, leading to problems known as Time Of Check to Time Of Use (TOCTOU) attacks [185, 59].

6.2.4 Practical Software Attestation

It is well-known that an unrestricted prover $P$ without trusted (hardware) components can always simulate the attestation algorithm $\text{Attest}$ and make the verifier $V$ accept. Hence, software attestation leverages side-channel information (typically the time $\delta$ that $P$ takes to compute the attestation algorithm $\text{Attest}$) to detect such simulations. Ideally, $V$ could disable the Computing Engine (CE) of the prover and directly read and verify the software state $S$ of $P$. However, exposing CE and the Secondary Memory (SM) of $P$ to $V$ in such a way requires hardware extensions on $P$, which contradicts the goal of software attestation to work with no hardware modifications.

To use the computation time $\delta$ of $P$ ($\text{Attest}$) in practical scenarios, it is critical to use an implementation of $\text{Attest}$ that has an identifiable lower time bound $\delta$ for the particular prover platform $P$. This means that it should be hard to find any other implementation of $\text{Attest}$ that can be executed by $P$ in significantly less time than $\delta$. Furthermore, the communication time jitter $\delta_{\text{Jit}}$ between $V$ and $P$ is typically much higher than the time needed by the computing engine of $P$ to perform a few instructions. Hence, to ensure that $V$ can measure also slight changes to $\text{Attest}$ (that could be exploited by a malicious prover to lie about its state), $V$ needs to amplify the effect of such changes. The most promising approach to realize this in practice is designing $\text{Attest}$ as an iterative algorithm with a large number of rounds, which is one of the security parameters of software attestation. Further, since showing the optimality of complex implementations is a hard problem and since $P$ must run $\text{Attest}$ in a reasonable amount of time, it is paramount that the individual rounds of $\text{Attest}$ are simple and efficient. As a result, cryptographically secure hash functions and complex Pseudo-Random Number Generators (PRNGs) are not a viable option. Hence, several previous works deployed lightweight ad-hoc designs of compression functions and PRNGs; however, without analyzing the underlying requirements on these components and their interaction. In contrast, we identify concrete requirements and provide a detailed
6.3 Generic Software Attestation

In this section, we formalize the general approach to software attestation and its underlying assumptions. In particular, we provide a precise set of formal assumptions that previously were only informally discussed or implicitly defined. Since we focus on a framework that is as general as possible, some specific constructions of prior work currently remain outside the scope of this model (cf. Section 6.7).

6.3.1 Protocol Specification

Our generic time-based attestation scheme (Figure 6.2) follows the practical approach discussed in Section 6.2.4: The verifier $V$ sends a random challenge $(g_0, r_0)$ to the prover $P$, which must iteratively compute a response $r'_N$ depending on $P$’s state $\text{State}(P)$ and $(g_0, r_0)$. On receipt of $r'_N$, $V$ tries to recompute $r'_N$ using the good state $S$ as input and accepts only if the result matches $r'_N$ and $P$ responded within a certain time threshold $\delta$.

The main components of this attestation scheme are two deterministic algorithms: A memory address generator

$$\text{Gen} : \{0, 1\}^{l_g} \rightarrow \{0, 1\}^{l_g + l_a}$$

and a compression function

$$\text{CHK}^N : \{0, 1\}^{l_r + N l_s} \rightarrow \{0, 1\}^{l_r}$$

where $l_g$ is the bit length of the state of $\text{Gen}$, $l_a$ is the bit length of the memory addresses, $l_r$ is the bit length of the attestation response $r'_N$, and $N \in \mathbb{N}$ is the number of iterations of $\text{Attest}$. Note that $l_g$, $l_a$, $l_r$, and $N$ are security-critical parameters and we discuss their effect later in Section 6.4. Furthermore, we provide an extended and iterative definition

$$r'_0 \leftarrow r_0$$

for $i = 1, \ldots, N$ do

$(g_i, a_i) \leftarrow \text{Gen}(g_{i-1})$

$s_i \leftarrow \text{Read}(S, a_i)$

$r'_i \leftarrow \text{CHK}(r_{i-1}, s_i)$

endfor

$(g_0, r_0) \leftarrow \{0, 1\}^{l_g + l_r}$

for $i = 1, \ldots, N$ do

$(g_i, a_i) \leftarrow \text{Gen}(g_{i-1})$

$s_i \leftarrow \text{Read}(S, a_i)$

$r_i \leftarrow \text{CHK}(r_{i-1}, s_i)$

endfor

Accept iff $r'_N = r_N \wedge t' - t \leq \delta$
of \( \text{CHK}^N \). For some value \( r_0 \in \{0, 1\}^l \) and \( \bar{s} := (s_1, \ldots, s_N) \), we define \( r \leftarrow \text{CHK}^N(c, \bar{s}) \) as \( r_i := \text{CHK}(r_{i-1}, s_i) \) for \( i = 1, \ldots, N \). Further, the address generator Gen is composed of two functions Upd and Sel, where Upd : \( \{0, 1\}^l \rightarrow \{0, 1\}^l \) updates the state \( g \) of Gen and Sel : \( \{0, 1\}^l \rightarrow \{0, 1\}^l \) generates a random memory address \( a \) from \( g \).

The attestation scheme depicted in Figure 6.2 works as follows: The verifier \( V \) sends an attestation challenge \((g_0, r_0)\) to the prover \( P \), who iteratively generates a sequence of memory addresses \((a_1, \ldots, a_N)\) based on \( g_0 \) using Gen. For each \( i \in \{1, \ldots, N\} \), \( P \) reads the state entry \( s_i = \text{Read}(S, a_i) \) at address \( a_i \), and iteratively computes \( r'_i = \text{CHK}(r'_{i-1}, s_i) \) using \( r'_0 = r_0 \). Finally, \( P \) sends \( r'_N \) to \( V \). \( V \) executes exactly the same computations as \( P \) using the good state \( S \) as input and compares the final result with the value received from \( P \). Eventually, \( V \) accepts if \( r'_N = r_N \) and \( P \) responded in time \( \delta := N(\delta_{\text{gen}} + \delta_{\text{read}} + \delta_{\text{CHK}}) + \delta_{\text{Net}} \), where \( \delta_{\text{gen}}, \delta_{\text{read}}, \) and \( \delta_{\text{CHK}} \) are the time bounds for running Gen, Read, and CHK, respectively, on a genuine and honest prover, and \( \delta_{\text{Net}} \) is the time bound for transmitting all protocol messages between \( V \) and \( P \).

Note that the basic concept of the protocol in Figure 6.2, which is modeled by our generic scheme, and several instantiations for specific platforms can be found in the literature on software-attestation (cf. Section 6.7). However, we aim to abstract from the particularities of individual platforms and instead design and analyze a construction that is as generic as possible.

### 6.3.2 Design Criteria and Assumptions

Before we analyze the security of the generic attestation scheme, we first discuss the design criteria of the underlying algorithms and formally define the assumptions required later in the security analysis. Note that, although some of these assumptions have been informally discussed or implicitly made in prior work, they have never been formally specified and analyzed before.

#### System-level Assumptions

We assume that the size of Primary Memory (PM) is limited and that the content of a randomly selected memory block of PM can be guessed only with a certain probability. These assumptions are quite natural since fast memory such as CPU cache tends to be expensive, and programmers and compilers generally aim for good code density and data locality to reduce accesses to slower mass storage [94].

#### Size of the Primary Memory

The size and utilization of PM by the attestation protocol plays a fundamental role for assessing the optimality of Attest with respect to the resources used by the prover \( P \). Therefore, we assume that the size of PM is limited such that, at any point in time, \( P \) can store only those values in PM that are necessary to complete the current computation.

---

2 The verifier \( V \) does not need to recompute the response \( r_N \) of the prover \( P \) during the attestation protocol-run and instead can use a database of pre-computed attestation challenge/response pairs.
**ASSUMPTION 1: SIZE OF THE PRIMARY MEMORY.** Let $g$ be the state of Gen, $a$ be an address of the Secondary Memory (SM), $s$ be a memory word of SM and $r$ be the attestation response as defined in Figure 6.2. We define $l_g := |g|, l_a := |a|, l_s := |s|, l_r := |r|$. The Primary Memory can store exactly one quadruple $(g,a,s,r) \in \{0,1\}^{l_g+l_a+l_s+l_r}$ at a time.

**State Incompressibility** A crucial assumption for any software attestation scheme that is not explicitly made in most works on software attestation is that a malicious prover cannot derive the good state from PM, i.e., without reading the good state from SM, with better probability than guessing while running Attest. Some of the software attestation schemes in the literature implicitly address this issue by adding entropy to the prover state, e.g., by filling unused memory areas with random data that cannot be predicted by the adversary. Since our goal is a generic framework that abstracts from particular instantiations, we formalize this aspect in the following assumption:

**ASSUMPTION 2: STATE INCOMPRESSIBILITY.** Let $S$ denote a state as specified in Definition 1 and let $D_S$ be the probability distribution

$$D_S(x) := \Pr[x = s|a \leftarrow \{0,1\}^{l_a} \land s := \text{Read}(S,a)]$$

for any $x \in \{0,1\}^{l_s}$. $S$ is incompressible iff for any algorithm $\text{Alg}$ that can be executed by $P$ and that does not invoke the Read-command it holds that

$$\Pr[x = s|a \leftarrow \{0,1\}^{l_a} \land x \leftarrow \text{Alg}(a) \land s := \text{Read}(S,a) \land \text{Time}_P(\text{Alg}) \leq \delta_{\text{Read}}] \leq \max_{x \in \{0,1\}^{l_s}} D_S(x) := \gamma.$$ 

**Computational Assumptions**

Since the security of software-based attestation relies on the computational and memory constraints of the prover’s hardware, the address generator Gen and the checksum CHK of the attestation algorithm Attest should be close to optimal with regard to the time required by the prover $P$ to execute them so that it is “hard” to find an alternative instantiation that can be executed by $P$ in less time. To formalize this aspect, we introduce the notion of (minimum) effort:

**Definition 4: Minimum Effort.** Let $\text{Alg}$ be a deterministic algorithm that can be executed by the prover $P$ and that accepts inputs $x$ from some domain $X$ and outputs $y$ from some domain $Y$. Further, we denote with $\theta_P(\text{Alg})$ the amount of a certain resource (e.g., time) required by $P$ to run $\text{Alg}$ for any input $x \in X$. For any other algorithm $\text{Alg}'$, we write $\text{Alg} \equiv \text{Alg}'$ iff $\text{Alg}(x) = \text{Alg}'(x)$ for all $x \in X$. That is, $\text{Alg} \equiv \text{Alg}'$ iff both algorithms $\text{Alg}$ and $\text{Alg}'$ realize exactly the same functionality. The minimum effort $\mu_P(\text{Alg})$ is the minimum $\theta_P$ required by $P$ to compute the results of $\text{Alg}$, i.e.,

$$\mu_P(\text{Alg}) := \min_{\text{Alg}' : \text{Alg} \equiv \text{Alg}'} \theta_P(\text{Alg'}).$$
This means that there is no algorithm \( \text{Alg}' \) that runs on \( \mathcal{P} \) and computes the result of \( \text{Alg} \) in less time than \( \mu_{\mathcal{P}}(\text{Alg}) \). Observe that \( \theta_{\mathcal{P}}(\text{Alg}) = \mu_{\mathcal{P}}(\text{Alg}) \) holds only if \( \text{Alg} \) is indeed an optimal implementation of the functionality provided by \( \text{Alg} \). We write \( \mu(\text{Alg}) \) instead of \( \mu_{\mathcal{P}}(\text{Alg}) \) in case the prover \( \mathcal{P} \) that is running \( \text{Alg} \) is clear from the context.

**Minimum Resource Requirement** We must assume that \( \text{Gen} \) and \( \text{CHK} \) and their implementations are designed such that any prover \( \mathcal{P} \) must at least invest a certain amount of resources, in our case the times \( \delta_{\text{Gen}} \) and \( \delta_{\text{CHK}} \), respectively, to gain any information on the outputs of \( \text{Gen} \) and \( \text{CHK} \). Without this assumption, \( \mathcal{P} \) could use an instantiation of \( \text{Gen} \) and \( \text{CHK} \) that can be executed in less time than \( \delta_{\text{Gen}} \) and \( \delta_{\text{CHK}} \), respectively. This would give \( \mathcal{P} \) time to perform additional computations that could be used to compute the correct attestation response \( r_N \) in time \( \delta \). We formalize this assumption as follows:

**Assumption 3: Time Requirement on \( \text{Gen} \).**

For all \( g \in \{0,1\}^{|y|} \) and \((g',a') := \text{Gen}(g)\), the prover \( \mathcal{P} \) requires at least time \( \delta_{\text{Gen}} := \mu_{\mathcal{P}}(\text{Gen}) \) to compute \( a' \) from \( g \). All algorithms that use less time than \( \delta_{\text{Gen}} \) can determine \( a' \) only with probability \( \max \{2^{-l_y}, 2^{-l_a}\} \). That is, for all algorithms \( \text{Alg} \) that can be executed by \( \mathcal{P} \) and all \( g \in \{0,1\}^{|y|} \) it holds that

\[
\Pr[\tilde{a} = a'|\tilde{a} \leftarrow \text{Alg}(g) \wedge (g',a') \leftarrow \text{Gen}(g) \wedge \text{Time}_{\mathcal{P}}(\text{Alg}) < \delta_{\text{Gen}}] \leq \max \{2^{-l_y}, 2^{-l_a}\}.
\]

Since \( \text{Gen} \) is the composition of \( \text{Upd} : \{0,1\}^{|y|} \rightarrow \{0,1\}^{|y|} \) and \( \text{Sel} : \{0,1\}^{|y|} \rightarrow \{0,1\}^{|y|} \), it holds for all algorithms \( \text{Alg} \) that can be executed by \( \mathcal{P} \) and all \( g \in \{0,1\}^{|y|} \) that

\[
\Pr[\tilde{g} = g'|\tilde{g} \leftarrow \text{Alg}(g) \wedge g' \leftarrow \text{Upd}(g) \wedge \text{Time}_{\mathcal{P}}(\text{Alg}) < \delta_{\text{Upd}}] \leq 2^{-l_y}
\]

\[
\Pr[\tilde{a} = a'|\tilde{a} \leftarrow \text{Alg}(g) \wedge a' \leftarrow \text{Sel}(g) \wedge \text{Time}_{\mathcal{P}}(\text{Alg}) < \delta_{\text{Sel}}] \leq 2^{-l_a}
\]

where \( \delta_{\text{Upd}} := \mu_{\mathcal{P}}(\text{Upd}) \) and \( \delta_{\text{Sel}} := \mu_{\mathcal{P}}(\text{Sel}) \).

**Assumption 4: Time Requirement on \( \text{CHK}^N \).**

For all \( y \in \{0,1\}^{l_r+N|y|} \) and \( r := \text{CHK}^N(y) \), the prover \( \mathcal{P} \) requires at least time \( N \cdot \delta_{\text{CHK}} := N \cdot \mu_{\mathcal{P}}(\text{CHK}) \) to compute \( r \) from \( y \). All algorithms that use less time than \( N \cdot \delta_{\text{CHK}} \) can determine \( r \) only with probability \( 2^{-l_r} \). That is, for all algorithms \( \text{Alg} \) that can be executed by \( \mathcal{P} \) and all \( y \in \{0,1\}^{l_r+N|y|} \) it holds that

\[
\Pr[\tilde{r} = r|\tilde{r} \leftarrow \text{Alg}(y) \wedge r \leftarrow \text{CHK}^N(y) \wedge \text{Time}_{\mathcal{P}}(\text{Alg}) < N \cdot \delta_{\text{CHK}}] \leq 2^{-l_r}.
\]

**Remark 4: Atomicity of \( \text{Gen} \) and \( \text{CHK} \).** Observe that Assumptions 3 and 4 require \( \text{Gen} \) and \( \text{CHK} \) to be atomic, i.e., no information on their outputs is leaked unless the complete required computation time was spent. This means that the adversary can either obtain the full result of \( \text{Gen} \) (or \( \text{CHK} \)) by executing \( \text{Gen} \) (or \( \text{CHK} \)) or at most guess its output. We make this assumption to keep the security analysis of the generic attestation scheme tightly structured by excluding adversaries that only partly execute \( \text{Gen} \) (or \( \text{CHK} \)) to learn their intermediate results, which may increase the adversary’s probability of determining
the correct output. Our proof can be extended to capture such adversaries but this would drastically increase its complexity and make it hard to follow. Instead, we exemplary argue for the implementation of Gen and CHK in Section 6.5.5 that, if the prover does not execute Gen and CHK completely, the probability that the prover determines the correct results decreases exponentially with the number of iterations, thus enforcing a “practical” atomicity.

Cryptographic Assumptions

The cryptographic requirements on Gen and especially CHK have not been systematically analyzed and formally specified before. We model these components based on the standard cryptographic models of PRNGs and hash functions.

Blind 2nd Preimage Resistance of CHK. The purpose of the compression function CHK is to map the state $S$ of the prover $P$ to a small attestation response $r_N$, which reduces the amount of data to be sent from $P$ to the verifier $V$ and enables the required large number of iterations of Attest (cf. Section 6.2.4). A necessary security requirement on CHK is that it should be hard for a malicious prover to replace the correct inputs with other values that yield the same attestation response $r_N$ as the good state $S$. This is similar to the common notion of 2nd preimage resistance of cryptographic hash functions. However, due to the time bound of Attest it is sufficient that CHK fulfills only a much weaker form of 2nd preimage-resistance, since we need to consider only “blind” adversaries who (in contrast to the classical definition of 2nd preimage resistance) neither know nor can compute the correct response $r_N$ to the verifier’s challenge $(g_0, r_0)$ of the attestation protocol-run, where the attestation shall be forged.

Hence, we introduce the definition of blind 2nd preimage-resistance:

**Assumption 5: Blind 2nd Preimage Resistance.**

$\text{CHK}^N : \{0, 1\}^{l_r + N l_s} \rightarrow \{0, 1\}^{l_r}$ is $\omega$-blind 2nd preimage resistant. This means that for any $N \in \mathbb{N}$, for any probability distribution $D$ on $\{0, 1\}^{l_s}$, for any algorithm $\text{Alg}(y)$ that can be executed by $P$ and that does not compute $\text{CHK}^N(y)$ it holds that

$$\Pr \left[ \tilde{r} = r | n_0 \overset{U}{\leftarrow} \{0, 1\}^{l_r} \wedge s_i \overset{D}{\leftarrow} \{0, 1\}^{l_s} : i \in \{1, \ldots, N\} \right. \left. \wedge (\tilde{r}_0, \tilde{s}_1, \ldots, \tilde{s}_N) \leftarrow \text{Alg}(n_0, s_1, \ldots, s_N) \right.$$ \left. \wedge (\tilde{r}_0, \tilde{s}_1, \ldots, \tilde{s}_N) \neq (r_0, s_1, \ldots, s_N) \right.$$ \left. \wedge \tilde{r} := \text{CHK}^N(\tilde{r}_0, \tilde{s}_1, \ldots, \tilde{s}_N) \right.$$ \left. \wedge r := \text{CHK}^N(r_0, s_1, \ldots, s_N) \right] \leq \omega.$$

Pseudo-Randomness of Outputs of Gen. To prevent the adversary from pre-computing forged attestation responses for a bad state $\tilde{S}$, we must assume that the memory addresses

---

3 For an input $x$ to a hash function $H$, it should be hard to find a second input $x' \neq x$ such that $H(x) = H(x')$. Hence, a 2nd preimage resistant hash function could instantiate CHK. However, standard hash functions are often considered too complex for efficient software attestation.

4 The adversary may still obtain the attestation responses to other verifier challenges, e.g., by eavesdropping on other attestation protocol-runs or by pretending to be the verifier to the prover.
a_i generated by Gen are pseudo-random\(^5\). Specifically, we assume that the output of Gen is computationally indistinguishable from uniformly random values within a certain time bound \(t\), i.e., the time bound of Attest. Formally:

**Assumption 6: Pseudo-randomness of Gen.**

Let \(C_{Gen}(b)\) be an algorithm that in case \(b = 0\) returns \((a_1, \ldots, a_N) \overset{\text{Unif}}{\leftarrow} \{0,1\}^{N,l_b}\). If \(b = 1\), then \(C_{Gen}(b)\) samples \(g_0 \overset{\text{Unif}}{\leftarrow} \{0,1\}^{l_g}\) and computes \((g_i, a_i) = Gen(g_{i-1})\) for \(i = 1\) to \(N\). In both cases \(C_{Gen}(b)\) outputs \((a_1, \ldots, a_N)\). Further, let \(Alg\) be an algorithm that on input \((a_1, \ldots, a_N)\) returns a bit \(b'\) as a guess for \(b\) used by \(C_{Gen}(b)\).

\[Gen : \{0,1\}^{l_g} \rightarrow \{0,1\}^{l_g + l_a} \text{ is } (t, \varrho)\text{-secure, i.e., for any algorithm } Alg \text{ that can be}
\]

executed by \(P\) in \(\text{Time}(Alg) \leq t\) it holds that

\[
\left| \Pr \left[ 1 \leftarrow Alg(a_1, \ldots, a_N) \mid (a_1, \ldots, a_N) \leftarrow C_{Gen}(0) \right] - \Pr \left[ 1 \leftarrow Alg(a_1, \ldots, a_N) \mid (a_1, \ldots, a_N) \leftarrow C_{Gen}(1) \right] \right| \leq \varrho.
\]

### 6.3.3 Security of the Generic Scheme

We now come to our main result, which we formulate in the following theorem that gives an upper bound for the success probability \(\varepsilon\) of any adversary against the generic software attestation scheme. Note that the term for \(\varepsilon\) is rather complex and will be derived step by step in the detailed security proof of Theorem 1 in Section 6.4.

**Theorem 1: Security of the Generic Scheme.**

Let \(q\) be the state of Gen, \(a\) be an address of SM, \(s\) be a memory word of SM and \(r\) be the attestation response as defined in Figure 6.2. Let \(l_g := |g|\), \(l_a := |a|\), \(l_s := |s|\), and \(l_r := |r|\). Let \(\Sigma \subset \{0,1\}^{l_g + l_a}\) denote an arbitrary set of good states (Definition 2) that comprise the generic software attestation scheme in Figure 6.2, let \(\tilde{S} := \text{State}(\tilde{P})\) be the state of the prover \(\tilde{P}\), let

\[\gamma_{\Sigma} := \max_{S \in \Sigma} \gamma = \max_{x \in \{0,1\}^{l_g + l_a}} D_{\tilde{S}}(x),\]

denoting the maximum probability of guessing a memory word \(s\) of a good state \(S \in \Sigma\), and let

\[\lambda_{\Sigma} := \max_{S \in \Sigma} \left| \left\{ a \in \{0,1\}^{l_a} \mid \text{Read}(\tilde{S}, a) = \text{Read}(S, a) \right\} \right| \cdot 2^{-l_a},\]

denoting the maximum probability that \(\tilde{S}\) matches one of the good states \(S \in \Sigma\) at some address \(a\). Further, let \(q\) (resp. \(k\)) be the number of instructions the computing engine of the prover \(\tilde{P}\) can perform in time \(\delta_{Gen} + \delta_{Read}\) (resp. \(\delta_{Read}\)). The attestation scheme in Figure 6.2 is an \(\varepsilon\)-secure attestation scheme with respect to \(\Sigma\) (Definition 3) if Assumptions 1 to 4 are fulfilled, CHK\(N\) is \(\omega\)-blind \(2^nd\) preimage resistant (Assumption 5) and Gen

---

\(^5\)Observe that unpredictability may be sufficient here. However, pseudo-randomness is currently required to prove the generic scheme.
6.4 Security Proof of the Generic Scheme

In this section, we derive the upper bound of the success probability $\varepsilon$ of any adversary against the generic software attestation scheme in Figure 6.2. We prove Theorem 1 using a hybrid argument, which is a common technique in cryptography. Hybrid arguments are particularly helpful to assess the security of protocols that comprise several security-relevant components with different properties and tasks. The idea is to start with an idealized version of the protocol, where all security-critical tasks are honestly executed. Technically, this is captured by introducing a hypothetical trusted third party (TTP) $T$ that initially performs all security-critical computations of the prover $P$. The idealized protocol is depicted in Figure 6.3.

The first step of the security analysis is to show the security of the idealized protocol. This step already reveals whether the final protocol can be sound and provide secure software attestation. Afterwards, the idealized protocol is transformed step by step, by moving security critical tasks from $T$ to $P$. For each transformation, we show based on the assumptions in Section 6.3.2 how the success probability of the adversary changes. The transformations are continued until $T$ becomes redundant in the last step (since all security critical tasks are then executed by $P$) and we arrive at the protocol in Figure 6.2. Note that $T$ only formally represents the honest execution of certain security-critical tasks and should not be confused with a real party. Consequently, we assume secure communication between $V$ and $T$ and that the time needed to transfer messages between $P$ and $T$ is zero.

6.4.1 Step 0: The Idealized Protocol

We first consider the idealized protocol depicted in Figure 6.3, where $T$ performs all security-critical computations of the prover $\bar{P}$ except $\text{Read}$. More detailed, verifier $V$ sends a random attestation challenge $r_0$ to $T$, which then generates a uniform sequence of addresses $(a_1, \ldots, a_N)$. Next, $T$ iteratively queries $\bar{P}$ with $a_1, \ldots, a_N$ and $\bar{P}$ should respond with its state entries $\tilde{s}_1, \ldots, \tilde{s}_N$ at the queried address. In each iteration, $T$ checks whether $\bar{P}$'s response $\tilde{s}_i$ equals the state entry $s_i$ of the good state $S$ and that $\bar{P}$
responded in time $\delta_{\text{Read}} := \mu(\text{Read})$. When all checks are successful, $\mathcal{T}$ sends the correct attestation response $r'_N = r_N$ to $\mathcal{V}$, otherwise some value $r'_N \neq r_N$. Recall that $\mathcal{V}$ accepts only if $r'_N = r_N$.

We are interested in the probability $\pi_0$ that $\mathcal{V}$ accepts a prover $\tilde{P}$ with $\text{State}(\tilde{P}) \neq S$ in the protocol in Figure 6.3. Since we do not know what $\tilde{P}$ is doing internally, we denote with $\tilde{x}_i$ its response to query $a_i$. We denote with round $i$ the timeframe between the point in time where $\tilde{P}$ receives $a_i$ and the point in time where $\tilde{P}$ receives $a_{i+1}$ for $i \in \{1, \ldots, N-1\}$. Moreover, with round $N$ we denote the timeframe between the point in time where $\tilde{P}$ receives $a_N$ and the point in time where $\tilde{P}$ sends the last protocol message to $\mathcal{T}$. Note that $\mathcal{T}$ ensures that $r'_N \neq r_N$ in case there is a round $i$ where $\tilde{P}$ responded with $\tilde{x}_i \neq s_i$ and/or used more time than $\delta_{\text{Read}}$ to respond to $a_i$. Hence, we consider only provers that take at most time $\delta_{\text{Read}}$ in each round.

Since all $a_i$ are sampled independently and since in each round $\tilde{P}$ can store only a tuple $(a, \tilde{x})$ in PM (Assumption 1), $\tilde{P}$ cannot use any information from round $i$ in another round $j \neq i$. Hence, the probability $p$ that $\tilde{P}$ responds to some $a_i$ with $\tilde{x}_i = s_i$ within time $\delta_{\text{Read}}$ is independent of all $a_j$ with $j \neq i$ and it holds that $\pi_0 = p^N$. It remains to estimate $p := \Pr[\tilde{x}_i = s_i]$ for some $i \in \{1, \ldots, N\}$. Let $R_i$ be the event that $\tilde{x}_i = \tilde{s}_i := \text{Read}(\tilde{S}, a_i)$,
i.e., \( \tilde{P} \) generated \( \tilde{x}_i \) by returning the entry of its state \( \tilde{S} \) at address \( a_i \). Then

\[
p = \Pr[\tilde{x}_i = s_i|R_i] \cdot \Pr[R_i] + \Pr[\tilde{x}_i = s_i|\neg R_i] \cdot \Pr[\neg R_i]
\leq \max \left\{ \Pr[\tilde{x}_i = s_i|R_i], \Pr[\tilde{x}_i = s_i|\neg R_i] \right\}.
\]

We now estimate these probabilities. Observe that

\[
\Pr[\tilde{x}_i = s_i|R_i] = \Pr[\tilde{s}_i = s_i|\tilde{s}_i := \text{Read}(\tilde{S}, a) \land s_i := \text{Read}(S, a) \land a \leftarrow \{0,1\}^{l_a}].
\]

We define

\[
\lambda := \left| \{ a \in \{0,1\}^{l_a} | \text{Read}(\tilde{S}, a) = \text{Read}(S, a) \} \right| \cdot 2^{-l_a}
\]

being the probability that \( \tilde{S} \) matches the good state \( S \) at some address \( a \). It holds that

\[
\Pr[\tilde{x}_i = s_i|R_i] = \lambda.
\]

It remains to estimate \( \Pr[\tilde{x}_i = s_i|\neg R_i] \), i.e., the probability that \( \tilde{x}_i = s_i \), where \( \tilde{x}_i \) has not been computed using \( \text{Read}(\tilde{S}, a_i) \). We distinguish between two cases. In the first case, \( \tilde{x}_i = \text{Read}(\tilde{S}, a_i) \) for some \( a_i \neq a_i \). That is the response given by \( \tilde{P} \) is indeed part of \( \tilde{S} \), but resides at a different address \( \tilde{a}_i \neq a_i \). Due to the tight time bound \( \delta_{\text{read}} \) in each round, \( \tilde{P} \) can perform just one single \( \text{Read} \) operation and does not have the time to derive \( \tilde{a}_i \) from \( a_i \). Thus, \( \tilde{a}_i \) and \( a_i \) are independent and it follows that

\[
\Pr[\tilde{s}_i = s_i|\tilde{s}_i := \text{Read}(\tilde{S}, \tilde{a}_i) \land s_i := \text{Read}(S, a) \} \leq \max_{x \in \{0,1\}^{l_a}} \mathbb{D}_{S}(x) := \gamma.
\]

In the second case, \( \tilde{x}_i \) is not the result of \( \text{Read} \), but computed by \( \tilde{P} \) in some other way. Since \( S \) is incompressible (Assumption 2), the probability that \( \tilde{x}_i = \text{Read}(S, a_i) \) is then upper bounded by \( \gamma \) as well. Thus, we have \( \Pr[\tilde{x}_i = s_i|\neg R_i] \leq \gamma \) and it follows that

\[
\pi_0 = \pi_0(N) \leq (\max \{ \lambda, \gamma \})^N.
\]

### 6.4.2 Step 1: Removing Check for Equality

Next we consider a protocol where \( T \) no longer verifies whether the responses \( \tilde{s}_i \) of \( \tilde{P} \) match \( s_i = \text{Read}(S, a_i) \) for \( i \in \{1, \ldots, N\} \) but instead uses \( \tilde{x}_1, \ldots, \tilde{x}_N \) as inputs to \( \text{CHK} \) to compute \( \tilde{r}_N := \text{CHK}^N(r_0, \tilde{x}_1, \ldots, \tilde{x}_N) \). However, \( T \) still checks whether \( P \) responds to each query \( a_i \) within time \( \delta_{\text{read}} \).

We estimate the probability \( \pi_1 \) that \( V \) accepts a prover \( \tilde{P} \) with \( \text{State}(\tilde{S}) \neq S \). By definition, \( T \) ensures that \( V \) rejects if any of the rounds takes more time than \( \delta_{\text{read}} \). Hence, to estimate an upper bound for \( \pi_1 \), we can restrict to provers \( \tilde{P} \) which respond in time \( \delta_{\text{read}} \) in each round. Let \( E \) denote the event that \( \tilde{P} \) returned \( \tilde{x}_i = s_i \) for all \( i \in \{1, \ldots, N\} \). Then

\[
\pi_1 = \Pr[\tilde{r}_N = r_N] \leq \max \left\{ \Pr[\tilde{r}_N = r_N | E], \Pr[\tilde{r}_N = r_N | \neg E] \right\}.
\]

Observe that \( \Pr[\tilde{r}_N = r_N | E] = \pi_0(N) \). Further, event \( \neg E \) implies that there is an \( \tilde{x}_i \neq s_i \).
In this case, it follows from Assumption 5 that $\tilde{r}_N = r_N$ only with probability $\omega$. Hence,

$$\pi_1 = \pi_1(N) \leq \max \{\omega, \pi_0(N)\}.$$ 

6.4.3 Step 2: Prover Controls Address Generation Time

Now, we consider a variant of the protocol of the previous step, where $\mathcal{P}$ can decide when it receives the next address $a_i$ from $\mathcal{T}$. This reflects the fact that in the final protocol, a malicious prover $\mathcal{P}$ may generate the memory addresses on its own whenever it wants to. Formally, this is captured by introducing a `req` protocol message which $\mathcal{P}$ needs to send to $\mathcal{T}$ for receiving the next address $a_i$. More precisely, when $\mathcal{P}$ sends the $i$-th request `req` to $\mathcal{T}$, $\mathcal{P}$ uniformly samples and sends the next address $a_i$ to $\mathcal{P}$. Note that $\mathcal{P}$ may request as many addresses as it wants and is not required to pose exactly $N$ requests. However, $\mathcal{T}$ ensures that $\mathcal{P}$ cannot change the order of the memory addresses, i.e., it only sends $a_i$ to $\mathcal{P}$ after $a_{i-1}$ has been sent. As in the previous step, $\mathcal{T}$ computes $\tilde{r}_N$ based on the responses $\tilde{x}_i$ sent by $\mathcal{P}$. Hereby, $\mathcal{T}$ processes the values $\tilde{x}_i$ in the order they are sent by $\mathcal{P}$. Furthermore, $\mathcal{T}$ no longer checks whether $\mathcal{P}$ responded to each $a_i$ in time $\delta_{\text{read}}$. However, $\mathcal{T}$ verifies that it receives at least $N$ responses $\tilde{x}_i$ from $\mathcal{P}$ in time $N \cdot \delta_{\text{read}}$. If this is not the case, $\mathcal{T}$ returns $r'_N \neq r_N$ to $\mathcal{V}$. Recall that, since $\mathcal{T}$ is a hypothetical party, sending `req` and receiving $a_i$ requires no time.

We now estimate the probability $\pi_2$ that $\mathcal{V}$ accepts a prover $\mathcal{P}$ with $\text{State}(\mathcal{P}) \neq S$. Observe that a prover that requests more than $N$ addresses $a_i$ from $\mathcal{T}$ does not gain any advantage over a prover that requests only up to $N$ addresses since $\mathcal{T}$ independently samples all $a_i$ and only the responses to the first $N$ addresses are used by $\mathcal{T}$ to compute $\tilde{r}_N$. Hence, we can restrict to provers $\mathcal{P}$ that request at most $N$ addresses $a_i$. Note that $\mathcal{P}$ must return values $\tilde{x}_1, \ldots, \tilde{x}_N$ such that $\text{CHK}^N(r_0, \tilde{x}_1, \ldots, \tilde{x}_N) = \text{CHK}^N(r_0, s_1, \ldots, s_N)$ and we have

$$\pi_2 = \Pr[\tilde{r}_N = r_N] \leq \max \{\Pr[\tilde{r}_N = r_N | \forall i : \tilde{x}_i = s_i], \Pr[\tilde{r}_N = r_N | \exists i : \tilde{x}_i \neq s_i]\}.$$ 

Note that in case $(\tilde{x}_1, \ldots, \tilde{x}_N) \neq (s_1, \ldots, s_N)$ it follows from Assumption 5 that $r'_N = r_N$ only with probability $\omega$. Hence, $\Pr[r'_N = r_N | \exists i : \tilde{x}_i \neq s_i] \leq \omega$.

It remains to estimate $\Pr[r'_N = r_N | \forall i : \tilde{x}_i = s_i]$. Recall that all $a_i$ is sampled independently by $\mathcal{T}$ and not predictable by $\mathcal{P}$. Hence, $\mathcal{P}$ cannot generate and store any information in round $i$ that may help in any later round $j > i$ to increase its success probability. However, $\mathcal{P}$ controls the time when the next $a_i$ is received and the time he uses in each round. Hence we can distinguish three different cases: (1) $\mathcal{P}$ uses exactly time $\delta_{\text{read}}$ in round $i$; (2) $\mathcal{P}$ uses less time than $\delta_{\text{read}}$ in round $i$; or (3) $\mathcal{P}$ uses more time than $\delta_{\text{read}}$ in round $i$. We denote with $p_e$ the upper bound of the probability that $\mathcal{P}$ returns $\tilde{x}_i = s_i$ in time $\delta_{\text{read}}$ after the point in time where $\mathcal{P}$ received $a_i$. Further, we denote with $p_i$ (resp. $p_m$) the upper bound of the probability that $\mathcal{P}$ returns $\tilde{x}_i = s_i$ in less (resp. more) time than $\delta_{\text{read}}$ after the point in time where $\mathcal{P}$ received $a_i$. It always holds $p_m \leq 1$. If $\mathcal{P}$ responds in less time than $\delta_{\text{read}}$, it follows from Assumption 2 that $\tilde{x}_i = s_i$ only with probability $p_i \leq \gamma$. Further, in case $\mathcal{P}$ uses exactly time $\delta_{\text{read}}$ in round $i$, then $\tilde{x}_i = s_i$ only with probability $p_e = \pi_0(1) \leq \max \{\lambda, \gamma\}$ (cf. Step 0). Now we consider the probability
over all $N$ rounds. Let $N_v$ be the number of rounds where $\overline{P}$ responds in exactly time $\delta_{\text{read}}$. Further, let $N_i$ (resp. $N_m$) be the number of rounds where $\overline{P}$ responds in time less (resp. more) than $\delta_{\text{read}}$, so that $N = N_v + N_i + N_m$. Let $k$ be the number of instructions that can be executed by the computing engine of $\overline{P}$ in time $\delta_{\text{read}}$ (cf. Section 6.2.2). Since we are interested in an upper bound of the success probability of $\overline{P}$, we make the following assumptions in favor of $\overline{P}$: (1) each round where $\overline{P}$ responds in less time than $\delta_{\text{read}}$ takes no time at all, i.e., $\overline{P}$ saves $k$ instructions in this round, and (2) for each of these rounds, the $k$ saved instructions can be used in $k$ other rounds. This implies $N_m = k \cdot N_i$ and thus
\[
\Pr [r_N' = r_N | \forall i : \overline{x}_i = s_i] \leq p_v^{N_v} \cdot p_i^{N_i} \cdot p_m^{N_m} \\
\leq p_v^{N_v} \cdot p_i^{N_i} \cdot 1^{k \cdot N_i} \\
\leq p_v^{N-(k+1) \cdot N_i} \cdot p_i^{N_i}.
\]
Since $p_v^{N-(k+1) \cdot N_i} = \pi_1(N - (k + 1)N_i)$ it follows that
\[
\pi_2 \leq \max \left\{ \max_{N_i : 0 \leq N_i \leq N} \{ \pi_1(N - (k + 1)N_i) \} : \lambda^{N_i} \right\}.
\]

6.4.4 Step 3: Moving Checksum Computation to Prover

In this step we consider the case where the prover $\overline{P}$ computes $\text{CHK}$ on its own and $T$ only checks whether $\overline{P}$ responds with some value $\tilde{r}_N$ in time $N(\delta_{\text{read}} + \delta_{\text{CHR}})$.

We are interested in the probability $\pi_3$ that the verifier $V$ accepts a prover $\overline{P}$ with $\text{State}(\overline{P}) \neq S$. Let $C$ be the event that $\overline{P}$ performs $\text{CHK}^N$, i.e., $\overline{P}$ computes $\tilde{r}_i = \text{CHK}(\tilde{r}_{i-1}, \tilde{x}_i)$ for $i = 1$ to $N$ using $\tilde{r}_0 = r_0$. Then we have
\[
\pi_3 = \Pr [\tilde{r}_N = r_N] \leq \max \{ \Pr [\tilde{r}_N = r_N | C], \Pr [\tilde{r}_N = r_N | \neg C] \}.
\]
Since event $\neg C$ means that $\overline{P}$ does not perform $\text{CHK}^N$, it follows from Assumption 4 that $\Pr [\tilde{r}_N = r_N | \neg C] \leq 2^{-l_r}$. Next we consider $\Pr [\tilde{r}_N = r_N | C]$. Event $C$ implies that $\overline{P}$ uses time $N \cdot \delta_{\text{CHR}}$ to compute $\text{CHK}^N$ and has only time $N \cdot \delta_{\text{read}}$ left to find $\tilde{x}_1, \ldots, \tilde{x}_N$ such that $\text{CHK}^N(\tilde{x}_1, \ldots, \tilde{x}_N) = \text{CHK}^N(s_1, \ldots, s_N)$. This is the same situation as in Step 2 so that $\Pr [\tilde{r}_N = r_N | C] \leq \pi_2$ and
\[
\pi_3 \leq \max \{ 2^{-l_r}, \pi_2 \}.
\]

6.4.5 Step 4: Replacing Random Sampling with Gen

Now we consider the case where $(a_1, \ldots, a_N)$ are generated by $\text{Gen}$ instead of being randomly sampled. We show that, due to Assumption 6, this modification does not significantly change the probability that $V$ accepts. To this end, we use $\overline{P}$ to construct an algorithm $D = D_{\overline{P}}$ that on input $(a_1, \ldots, a_N)$ aims to distinguish between the two cases $(a_1, \ldots, a_N) \leftarrow \text{Gen}(0)$ and $(a_1, \ldots, a_N) \leftarrow \text{Gen}(1)$. We show that, if the success probability of $\overline{P}$ significantly changes, this directly violates Assumption 6. $D$ works as follows: $D$ emulates $T$ and sequentially sends $(a_1, \ldots, a_N)$ to $\overline{P}$, which eventually returns $\tilde{r}_N$. Then, $D$ sends $\tilde{r}_N$ to $V$, which either accepts or rejects. $D$ outputs 1 if $V$ accepts and 0 otherwise.
Note that in case of $C_{\text{Gen}}(0)$ the addresses $(a_1, \ldots, a_N)$ are randomly sampled as in the protocol considered in Step 3, which means that $\tilde{P}$ will succeed with probability $\pi_3$. In case of $C_{\text{Gen}}(1)$ the addresses are generated using Gen and $P$ may succeed with some success probability $\pi_4$. Furthermore, by the definition of $D$ it holds that

$$
\begin{align*}
\Pr[1 \leftarrow D(a_1, \ldots, a_N)|(a_1, \ldots, a_N) \leftarrow C_{\text{Gen}}(0)] &= \pi_3 \\
\Pr[1 \leftarrow D(a_1, \ldots, a_N)|(a_1, \ldots, a_N) \leftarrow C_{\text{Gen}}(1)] &= \pi_4.
\end{align*}
$$

Since Gen is assumed to be $(N \cdot (\delta_{\text{read}} + \delta_{\text{chk}}), \varrho)$-secure (Theorem 1), it follows from Assumption 6 that

$$
\varrho \geq |\Pr[1 \leftarrow D(a_1, \ldots, a_N)|(a_1, \ldots, a_N) \leftarrow C_{\text{Gen}}(0)] - \Pr[1 \leftarrow D(a_1, \ldots, a_N)|(a_1, \ldots, a_N) \leftarrow C_{\text{Gen}}(1)]| = |\pi_3 - \pi_4|
$$

and hence

$$
\pi_4 \leq \pi_3 + \varrho.
$$

### 6.4.6 Step 5: Transition to Final Protocol

In the last step, we consider the final protocol depicted in Figure 6.2, where $a_1, \ldots, a_N$ are generated by $V$ and $P$ using Gen. Note that $T$ is removed completely since its functionality is reduced to relaying protocol messages between $V$ and $P$.

We estimate the probability $\pi_5$ that $V$ accepts a malicious prover $\tilde{P}$ with $\text{State}(\tilde{P}) \neq S$. Again, let $C$ be the event that $\tilde{P}$ performs $\text{CHK}^N$ (as in Step 3). Then

$$
\pi_5 = \Pr[\tilde{r}_N = r_N] \leq \max \left\{ \Pr[\tilde{r}_N = r_N|C], \Pr[\tilde{r}_N = r_N|\neg C] \right\}.
$$

In case $\tilde{P}$ does not perform $\text{CHK}^N$ (event $\neg C$), we have $\Pr[\tilde{r}_N = r_N|C] \leq 2^{-l_r}$ due to Assumption 4. Now consider the event $C$ where $\tilde{P}$ computes $\text{CHK}^N$. Observe that $\tilde{P}$ must determine $(\tilde{r}_0, \tilde{x}_1, \ldots, \tilde{x}_N)$ such that $\tilde{r}_N = \text{CHK}(\tilde{r}_0, \tilde{x}_1, \ldots, \tilde{x}_N) = \text{CHK}(r_0, s_1, \ldots, s_N) = r_N$.

We first consider the case where $\tilde{P}$ cannot determine at least one state entry $s_i$, which means that $(\tilde{r}_0, \tilde{x}_1, \ldots, \tilde{x}_N) \neq (r_0, s_1, \ldots, s_N)$. Clearly, in this case $\tilde{P}$ can neither compute $r_i = \text{CHK}(r_{i-1}, s_i)$ nor $r_N = \text{CHK}^N(r_0, s_1, \ldots, s_N)$ and it follows from Assumption 5 that $\tilde{r}_N = r_N$ only with probability $\omega$. Next, we estimate the probability that $\tilde{P}$ determines all state entries

$$
s_i = \text{Read}(S, a_i) = \text{Read}(S, \text{Sel}(g_i)) = \text{Read}(S, \text{Sel}(\text{Upd}(g_{i-1})))
$$

for $i \in \{1, \ldots, N\}$. We denote with round $i$ the time frame between the point in time where $\tilde{P}$ stores $\tilde{r}_{i-1}$ in PM and the point in time where $\tilde{P}$ stores $\tilde{r}_i$ in PM. In principle, $\tilde{P}$ could pre-compute some values relevant for the $i$-th execution of $\text{CHK}$ in some round $j < i$. However, due to the limited size of PM (Assumption 1), $\tilde{P}$ can store only those values in PM that are needed for one particular round. Hence, $\tilde{P}$ can either (1) discard these pre-computed values and recompute them later when needed, or (2) store the pre-
computed values in SM and read them from SM later, when they are needed in round 
i. Clearly, both cases induce a time overhead compared to an attack where \( \tilde{P} \) computes these values only when needed, i.e., in the \( i \)-th round before the \( i \)-th execution of CHK.
Hence, to estimate an upper bound for \( \Pr \left[ \tilde{r}_N = r_N | \neg C \right] \) we can restrict to provers that compute all \( g_i \) and \( a_i \) in the right order, i.e., that compute \( \tilde{g}_i = \text{Upd}(\tilde{g}_{i-1}) \) and \( \tilde{a}_i = \text{Sel}(\tilde{g}_i) \)
for \( i = 1 \) to \( N \) with \( \tilde{g}_0 = g_0 \). Let \( G \) be the event that \( \tilde{P} \) computes \( \tilde{g}_i = \text{Upd}(\tilde{g}_{i-1}) \) and \( \tilde{a}_i = \text{Sel}(\tilde{g}_i) \) for \( i = 1 \) to \( N \) with \( \tilde{g}_0 = g_0 \). Then

\[
\Pr \left[ \tilde{r}_N = r_N | \neg C \right] \leq \max \{ \Pr \left[ \tilde{r}_N = r_N | \neg C \wedge G \right], \Pr \left[ \tilde{r}_N = r_N | \neg C \wedge \neg G \right] \}.
\]

Note that event \( G \) implies that \( \tilde{P} \) used time \( N \cdot (\delta_{\text{upd}} + \delta_{\text{sel}}) \) to compute the correct addresses \( a_1, \ldots, a_N \). Moreover, since we consider only provers that compute \( g_i \) and \( a_i \) in the right order, event \( G \) also means that \( \tilde{P} \) cannot determine any information on \( a_j \) with \( j > i \) before computing \( a_i \). Hence, we are in the same situation as in Step 4 and \( \Pr \left[ \tilde{r}_N = r_N | \neg C \wedge G \right] = \pi_4 \).

Next we consider the event \( \neg G \), where there is at least one round where \( \tilde{P} \) did not perform \text{Upd} or \text{Sel}. Similar to Step 2 we consider three different types of rounds: (1) \( \tilde{P} \) performs \text{Upd}, \text{Sel} and \text{Read} in the required time; (2) \( \tilde{P} \) behaves as in the first case and performs some additional computations; and (3) \( \tilde{P} \) takes less time than in the first case, omitting \text{Upd}, \text{Sel} or \text{Read}.

Let \( q \) be the number of instructions that can be executed by the computing engine of \( \tilde{P} \) in time \( \delta_q := \delta_{\text{upd}} + \delta_{\text{sel}} + \delta_{\text{read}} \) (cf. Section 6.2.2). Further, we denote with \( p_e \), \( p_m \) and \( p_t \) the probability that \( \tilde{P} \) determines \( \tilde{x}_i = s_i \) in a round \( i \) that takes equal, more or less than time \( \delta_q \). We determine an upper bound of these probabilities by making two assumptions in favor of \( \tilde{P} \): (1) we assume that a round where \( \tilde{P} \) does not perform \text{Upd}, \text{Sel} or \text{Read} takes no time at all, i.e., \( \tilde{P} \) saves \( q \) instructions in this round, and (2) for each of these rounds, the \( q \) saved instructions can be used in \( q \) other rounds. We first estimate \( p_e \), where \( \tilde{P} \) behaves exactly like an honest prover. Hence, \( \tilde{x}_i = s_i \) only if \( \text{Read}(\tilde{S}, \text{Sel}(\text{Upd}(\tilde{g}_{i-1}))) = s_i \) and \( p_e \leq \lambda \). It always holds that \( p_m \leq 1 \). Now we estimate \( p_t \), where \( \tilde{P} \) omits \text{Upd}, \text{Sel} or \text{Read}. Note that in case \( \tilde{P} \) did not perform \text{Upd} (resp. \text{Sel}) in round \( i \), it follows from Assumption 3 that \( \tilde{g}_i = g_i \) (resp. \( \tilde{a}_i = a_i \)) only with probability \( 2^{-l_g} \) (resp. \( 2^{-l_a} \)). Moreover, in case \( \tilde{P} \) does not perform \text{Read}, \( \tilde{x}_i = s_i \) holds only with probability \( \gamma \) (Assumption 2). Hence, \( p_t \leq \max \{ 2^{-l_g}, 2^{-l_a}, \gamma \} \). We denote with \( N_e, N_m, N_t \) the number of rounds where \( \tilde{P} \) invested equal, more or less than time \( \delta_q \). Then we have \( N = N_e + N_m + N_t \) and \( N_m \leq q \cdot N_t \). Then

\[
\Pr \left[ \tilde{r}_N = r_N | \neg C \wedge \neg G \right] \leq p_e^{N_e} \cdot p_t^{N_t} \cdot p_m^{N_m} \\
\leq p_e^{N_e} \cdot p_t^{N_t} \cdot 1^{N_m} \\
\leq p_e^{N - (q+1)N_t} \cdot p_t^{N_t} \\
\leq \lambda^{N - (q+1)N_t} \cdot \left( \max \{ 2^{-l_g}, 2^{-l_a}, \gamma \} \right)^{N_t}.
\]
Thus, we have

\[ \pi_5 \leq \max \left\{ 2^{-l_r}, \pi_4, \max_{N_i \mid 0 \leq N_i \leq N} \left\{ \lambda^{N - (q+1)N_i} \cdot \left( \max \left\{ 2^{-l_q}, 2^{-l_a}, \gamma \right\} \right)^{N_i} \right\} \right\}. \]

### 6.4.7 The Impact of Network Jitter

In the final protocol, the time \( \delta_{\text{Net}} \) required for transmitting the protocol messages between verifier \( V \) and prover \( P \) must be considered. In practice, the transmission time \( \delta_{\text{Net}}' \) is subject to network jitter and differs between protocol runs. Hence, \( \delta_{\text{Net}} \) is an upper bound for \( \delta_{\text{Net}}' \) that should be determined by statistical analysis. Note that a malicious prover \( P \) could exploit \( \delta_{\text{Jit}} := \delta_{\text{Net}} - \delta_{\text{Net}}' \) to perform additional operations when computing \( \tilde{r}_N \). Hence, it must be ensured that no prover can compute \( \tilde{r}_N = r_N \) in time \( \delta + \delta_{\text{Jit}} \). As shown in the security proof, \( r_N \) must be computed sequentially and each intermediate \( r_i \) is needed. Assume that \( P \) aims to use \( \delta_{\text{Jit}} \) to perform additional computations in some round \( i \). Then \( P \) must at least perform one additional operation in each round to identify round \( i \). Hence, to prevent \( \tilde{P} \) from exploiting \( \delta_{\text{Jit}} \), the number of rounds \( N \) must be set such that \( \delta_{\text{Jit}}/N \) is less than the time needed for the most basic operation that can be performed by the Computing Engine (CE) of an honest prover, i.e., one clock cycle.

### 6.5 Instantiation of Generic Attestation Scheme

In the following, we discuss a concrete realization of the generic scheme in Figure 6.2 and how it fulfills the assumptions in Section 6.3.2. A major result of our analysis is that a secure software attestation algorithm can be designed using two similar Pseudo-Random Number Generators (PRNGs). We show an implementation of a secure address generator Gen and blind 2nd preimage resistant \( \text{CHK} \) function (Assumption 5) based on PRNGs and perform a detailed analysis of the optimality of the resulting program code.

As the target platform of our instantiation, we chose an Atmel ATtiny44A System on Chip (SoC). It is a popular modern representative of devices that target minimal power consumption and costs, and thus an ideal platform to validate our goal of providing attestation for low-cost embedded systems\(^6\).

The ATtiny44A features an Atmel AVR MCU with \( 32 \times 8 \) bit registers, 256 byte Static RAM (SRAM) and a slightly slower 4,096 byte program memory (PROM\(^7\)). The CPU supports basic arithmetic and boolean operations such as addition and rotation of registers, but no multiplication. For efficient memory addressing, 6 of the CPU registers can be treated as 3 16-bit registers in some operations. We argue that the following attestation algorithm is blind 2nd preimage resistant (Assumption 5) and time-sensitive

---

\( ^6 \)Note that the majority of low-cost computing platforms fit to our computational model in Section 6.2, and related work also proposed extensions for multi-core and other configurations [334, 192]

\( ^7 \)Although Programmable ROM (PROM) remains a common abbreviation, it is usually realized as writable Flash memory to allow software updates and may thus be compromised.
(Assumptions 3 and 4):

\[
\begin{align*}
\text{Gen} : g_i := \text{Upd}(g_{i-1}), \vec{a}_i := \text{Sel}(g_i) \\
\text{CHK} : \hat{r}_i := r_{i-1} \oplus \text{Read}'(\vec{a}_i), r_i := \text{Upd}(\hat{r}_i)
\end{align*}
\] (6.1) (6.2)

Here, \(\text{Sel}\) extracts multiple memory addresses \(
\vec{a}_i := (a_{i,1}, a_{i,2}, \ldots)\) with a specific data format from \(g_i\), \(\text{Upd}\) is the PRNG update function underlying \(\text{Gen}\) and also provides bit confusion in \(\text{CHK}\), and \(\vec{x}_i \leftarrow \text{Read}'(\vec{a}_i)\) is a generalized version of the atomic \(\text{Read}\) operation that takes \(\vec{a}_i\) and returns multiple memory blocks \(\vec{x}_i := (x_{i,1}, x_{i,2}, \ldots)\).

### 6.5.1 Practical Address Generators

Provably secure PRNGs are too complex and too slow for practical applications, especially in the case of low-cost embedded systems. Hence, practical constructions are usually analyzed using empirical tests and cryptanalysis, or build upon other cryptographic primitives such as block ciphers. In particular, since simplicity and speed is paramount in software attestation, previous works often resort to simple pseudo-random recursions and T-functions when realizing \(\text{Gen}\) for low-cost embedded systems [272, 271].

Indeed, our analysis shows that, unlike traditional cryptographic scenarios, we only require \(\text{Gen}\) to be secure within a very short time frame. Hence it is reasonable to concentrate on lightweight constructions which show good statistical properties, e.g., using the Diehard\(^8\) test suite. Several such lightweight constructions that are suitable for \(\text{Upd}\) have been presented and analyzed in prior work [141]. We select a PRNG instantiation that bijectively maps a \(4 \times 32\) bit PRNG state \(g_{i-1} := (A_{i-1}|B_{i-1}|C_{i-1}|D_{i-1})\) to \(g_i := (A_i|B_i|C_i|D_i)\) via

\[
\begin{align*}
A_i &:= A_{i-1} + \text{rotr}(D_{i-1}, 8) \\
B_i &:= B_{i-1} + \text{rotr}(A_i, 8) \\
C_i &:= C_{i-1} + \text{rotr}(B_i, 8) \\
D_i &:= D_{i-1} + \text{rotr}(C_i, 8),
\end{align*}
\]

where \(\text{rotr}(\cdot, 8)\) is the byte-wise right-rotation. According to the analysis conducted in [141] and our own tests, the length of the sequence of this PRNG is at least \(2^{32}\) in the overwhelming number of cases, and will thus exceed the number of rounds \(N\) commonly used in practical applications. Furthermore, this construction passed all Diehard tests, i.e. it has good statistical properties, and the verifier can verify the sequence length during his own simulation run during attestation. Hence, one may expect that a distinguisher (cf. Assumption 6) can only be successful if he computes first the initial state \(g_0\) from the outputs \(\vec{a}_i\), afterwards generates the outputs of \(\text{Gen}\) on his own, and eventually compares the results with the given values. However due to the optimality of \(\text{Gen}\) (Assumption 3 shown in Section 6.5.4), one can derive a lower time bound for this approach. If the time constraint is set below this bound, indistinguishability from random values is highly expected.

\(^8\)http://www.stat.fsu.edu/pub/diehard/
6.5.2 Primary and Secondary Memory

Next, we define and implement access to primary and secondary memory such that Assumptions 1 and 2 are met. For the realization of \texttt{Read} (Assumption 2), observe that most platforms including our target platform have a fixed, hardware-dictated time for accessing slower memory regions, and that the probability $\gamma = \mathbb{D}_S(x)$ of guessing the memory content at a specific memory address is a security parameter of the particular good state $S$. Hence it remains to specify what is considered as Secondary Memory (SM). Since our target platform follows the Harvard architecture, i.e., CPU instructions are fetched via a separate bus, the primary target of \texttt{Read} is the Programmable ROM. However, the ATtiny44A also includes a faster 256 byte SRAM memory, which could be used to cache PROM data and thus realize a partially faster \texttt{Read} operation. Hence we refine our notion of SM: The SM includes all memory types accessible to the Computing Engine (CE) that can be randomly sampled with equal or faster speed than the memory where CE fetches instructions from, excluding primary memory.

In our case, SM thus includes the PROM and the SRAM of the prover platform and we must refine \texttt{Read} to encompass both memory types. Specifically, we extend the prover algorithm to include two read functions \texttt{Read}\textsuperscript{(1)} and \texttt{Read}\textsuperscript{(2)} for accessing PROM and SRAM in time $\delta_{\text{Read}\textsuperscript{(1)}}$ and $\delta_{\text{Read}\textsuperscript{(2)}}$, respectively. Both memory types are sampled at the same frequency in every round, so that no advantage can be gained by exchanging state entries between slower and faster memory. This is consistent with our definition of \texttt{Read} in Section 6.2.2:

\begin{align*}
\vec{a}_i := (a^{(1)}_i, a^{(2)}_i) & \leftarrow \text{Gen}(g_{i-1}) \\
\vec{x}_i := (x^{(1)}_i, x^{(2)}_i) & \leftarrow \text{Read}'(\vec{a}_i).
\end{align*}

Observe that the content of these memories may have different probability distributions $\gamma^{(j)} = \mathbb{D}_{S(x)}(x)$ and different memory address lengths $l_{a^{(j)}}$ with $j \in \{1, 2\}$. This must be considered as follows when determining $\varepsilon$ in Theorem 1:

$$\gamma := \max \{\gamma^{(1)}, \gamma^{(2)}\}, \quad l_a := \max \{l_{a^{(1)}}, l_{a^{(2)}}\}.$$  

This approach can be generalized to arbitrary sets of memory comprising SM. In particular, all but the fastest available memory types of $\mathcal{P}$ can be turned into SM, since the pseudo-random access patterns of \texttt{Read} effectively disable any hardware pre-fetching and caching mechanisms [94]. Further, any additional (tertiary) memory in the system cannot be used by $\mathcal{P}$ to provide SM without increasing the average access time $\delta_{\text{Read}} = \frac{1}{2} (\delta_{\text{Read}\textsuperscript{(1)}} + \delta_{\text{Read}\textsuperscript{(2)}})$, so Assumption 2 is met.

We now consider Assumption 1. The ATtiny44A has a large amount of CPU registers and lacks Level-1 cache. Hence, the Primary Memory of this platform consists only of 32 8-bit registers. We implement the attestation algorithm using \texttt{Upd} with $l_g = 4 \times 32$ bit and a variation of \texttt{Upd} running on $l_r = 3 \times 32$ bits as a part of \texttt{CHK}. Two more registers are required to hold the loop counter $i$ of the algorithm and the last two remaining registers are used as temporary variables to store the 16 bit memory address $a_i$ and the corresponding memory block $s_i$. As shown in Lines 1–7 of Listing 6.1, no free registers remain to hold additional data, and hence Assumption 1 holds.
6.5. INSTITUTION OF GENERIC ATTESTATION SCHEME

In particular, due to our generalization of Read to encompass multiple memory accesses at different addresses, there is no point in time at which the CPU has immediate access to all of $\tilde{x}$ or $\tilde{a}$ of the current or previous rounds.

6.5.3 Designing the Checksum Function

We now show that our construction of $\text{CHK}^N$ (Equation 6.2) fulfills the 2nd preimage resistance property (Assumption 5). Recall that the goal of the adversary is to find a 2nd preimage $(\tilde{r}_0, \tilde{x}_1, \ldots, \tilde{x}_N)$ for an input $(r_0, s_1, \ldots, s_N)$ but the adversary cannot compute $r_N = \text{CHK}^N(r_0, s_1, \ldots, s_N)$. That is, there exists at least one index $i$ such that the adversary never executed $\text{CHK}(r_i, s_i)$. Observe that $\text{CHK}$ is bijective with respect to both inputs, i.e., fixing one input yields a bijection on the other input. Let $i$ denote the smallest index such that $\tilde{x}_i \neq s_i$. Since in this case $\tilde{r}_{i-1} = r_{i-1}$, it follows that $\tilde{r}_i = \text{CHK}(\tilde{r}_{i-1}, \tilde{x}_i) \neq \text{CHK}(r_{i-1}, s_i) = r_i$. Hence, there must be a second index $j > i$ such that $\tilde{x}_j \neq s_j$ corrects the error introduced by $\tilde{x}_i$. However, due to the time bound of the attestation scheme, the adversary never computed $r_i = \text{CHK}(r_{i-1}, s_i)$. Hence, in the best case the adversary may deduce from the known difference $s_i \oplus \tilde{x}_i$ some information on the difference $\tilde{r}_i \oplus r_i$.

As we show in the following, our checksum design statistically maximizes the difference between the outputs of $\text{CHK}$ even for minimal (i.e., single bit) changes to the inputs, minimizing the probability that the adversary can determine any information on the difference $\tilde{r}_{i+1} \oplus r_{i+1}$. Further, we show that the iterative structure of $\text{CHK}^N$ ensures that the probability that the adversary determines any information on any $\tilde{r}_j \oplus r_j$ for $j > i$ decreases significantly in each subsequent round. This property can be evaluated using the well-known avalanche criterion, which is an often used heuristic for measuring the bit confusion and diffusion properties of cryptographic algorithms including hash functions. The avalanche test measures the average probability of the individual output bits of a function to flip given that a single random input bit was flipped.

Evaluation of Checksum Sensitivity

We performed an avalanche test to determine the average bit-sensitivity of our checksum design and compared it to previously proposed checksums. For this purpose, we took several well-known schemes and replaced the PRNGs underlying their address generator with the Linux system PRNG, such that PRNG flaws can be excluded and the PRNG can be conveniently saved and reset. Moreover, we modified their Read function such that a random single bit error $e$ is induced on the first requested memory sample $\tilde{x}_1 := s_1 \oplus e$ used as input to $\text{CHK}^N$. The overall memory state $S$ was simulated as a static 8 KB buffer of arbitrary but fixed program code.

After running the attestation protocol once with $\tilde{x}_1$, we reset the PRNG seed and repeat the protocol using the original $s_1$ without error $e$. We repeated these tests 100,000 times with random initial PRNG seeds and random single-bit errors $e$. For each test, the bitwise difference ($\oplus$) in the checksum outputs $r_i$ was recorded for several rounds $i$. We then derived the bit sensitivity of $\text{CHK}$ at round $i$ as the average probability over all checksum bits to flip after $i$ rounds.
Evaluated Algorithms. While we aimed to evaluate the most representative instantiations of software attestation schemes, our selection is limited by the availability of functional code, since many details are difficult to discern from textual descriptions and the provided pseudo-code is often incomplete or ambiguous.

In this regard, we explicitly like to thank the authors of VIPER [192] for providing us with their original prototype implementation. VIPER follows a different design from most previous schemes, employing several timed nonces and intermediate checksum responses similar to Figure 6.3. To allow a consistent comparison, we treat the concatenation of these intermediate attestation responses as one large response $r_N$.

Apart from our own PRNG-based attestation algorithm presented in Section 6.5.1, we additionally selected the following algorithms based on their published pseudo-code:

- Block-based Pseudorandom Memory Traversal (BPMT) was proposed to reduce the required amount of memory block iterations by measuring multiple subsequent memory blocks at once [335]. Their scheme was reused and extended in [23].

- Indisputable Code Execution (ICE) was proposed for Secure Code Update By Attestation (SCUBA [272]) and reused later as a root of trust in key establishment [271]. Designed and implemented for a 16-bit MSP430 CPU, ICE uses a 16-bit T-function as PRNG.

- PRISM [113] was presented as a mechanism for user-verifiable platform attestation, with a prototype implementation for a 32-bit MIPS system using a 32-bit T-function as PRNG.

Checksum Sensitivity. Figure 6.4 shows the average sensitivity per checksum bit over several iterations $i$, with a maximum average chance of 50% per checksum bit to be flipped upon a single-bit input change. The results show that our PRNG-based checksum construction performs notably better than previously proposed constructions, which did not provide a systematic way to ensure strong bit confusion/diffusion in the checksum computation. In particular, the poor bit confusion/diffusion property of BPMT [335] is caused
by the author's proposed optimization to select multiple subsequent memory blocks based
on one single memory address $a_i$ and to iteratively merge them into the *same* checksum
position using the $\oplus$ operation. This issue is only partly mitigated the subsequent exten-
sion in [23], where the number of subsequently merged blocks is randomized.

In contrast, in our checksum construction a single bit flip incurs a large amount of
side-effects already after the first few rounds, and all checksum bits are flipped with
probability greater than 0.49 already after the 6th round. Hence, our method of using
the PRNG underlying the address generator Gen also for bit confusion/diffusion in the
checksum function CHK proved to be very effective.

6.5.4 Implementation and Optimality

We now present the low-level implementation of the checksum, address generator and
memory access functions and assess their implementation optimality according to As-
sumption 3 and Assumption 4.

While it is hard to show in general that an algorithm and its implementation are op-
timal, prior work usually highlights the limited code complexity to argue that an optimal
implementation is possible and will eventually be found. However, due to our systematic
algorithm design and the limited complexity of the Computing Engine (CE) of our target
platform, we achieve a high confidence that no alternative implementation of the code in
Listing 6.1 exists that uses less clock cycles and at the same time computes the correct
attestation response with probability greater than $\varepsilon$.

Listing 6.1 shows the detailed implementation of our prover algorithm. The $N$ rounds
are processed in batches of 3, where each batch performs Upd (Lines 8–24), extracts
$2 \times 3$ memory addresses $a_{i,j}$, and reads $2 \times 3$ memory samples $x_{i,j}$ from PROM and
SRAM (Lines 25–64). The resulting 6 memory samples are then merged twice, at different
positions of the 12 byte $r_i$, to increase the impact of manipulated memory blocks and
accelerate the subsequent avalanche effect. Subsequently, CHK is computed by applying
Upd to the checksum state $r_i$ for bit confusion and diffusion (Lines 65–77). Observe
that expanding the loop gives no advantage to the adversary since this would save only a
fraction of an instruction per round, while on the other hand instructions must be inserted
to hide the manipulated PROM areas.

In the following we analyze the implementation of each of the underlying components
(Gen, Read, CHK) and argue why the computing engine cannot execute them any faster.
The employed CPU instructions and their clock cycles are summarized in Table 6.1.

PRNG Optimality (Lines 8–24, 65–77)

The Upd operation used by Gen and CHK is reduced to adding up 32-bit words in-place, with
implicit rotation through byte swapping. As the inputs to the addition are pseudo-random
and only determined at runtime, only the class of add/sub instructions can perform this
operation efficiently. Among these, two instructions compute on a 16-bit value in 2 cycles,
while others compute on 8-bit values in 1 cycle. Hence, there is no combination of add/sub
instructions that takes less than 4 cycles to add two 32-bit integers as done in Listing 6.1.
Address Extraction (Lines 25–64)

For sampling the 4,096-byte PROM, we require 12 random bits in an otherwise zeroed 16-bit address register ($Z$). The bits should be copied from $g_i$ to avoid overhead of restoring $g_i$ ($\text{movw}$). The upper 4 bits of $Z$ can then be zeroed using bit masks, by resetting individual bits or by rotation/shifting. Taking only a single cycle, the bit mask ($\text{andi}$) is most efficient here. Hence, the PROM address extraction takes at minimum 2 cycles, as realized in Lines 26–27, 31–32 and 36–37.

For the SRAM memory, the required 8 pseudo-random bits should again first be copied from $g_i$ to avoid its modification, and another cycle is required to zero the higher order byte of $Z$. Moreover, an offset must be added to the memory addresses, which cannot be added efficiently using boolean operations and requires an addition/subtraction operation. In fact, a full 16-bit addition is required to handle potential overflows (carry) in about 1/3 of the cases. However, since the Computing Engine (CE) does not support the immediate addition/subtraction of values larger than 63, we must either perform two 16-bit additions (4 cycles) or store the SRAM offset in a temporary register (1 extra cycle). Hence an extra 3 cycles are at least required for a correct addition of the SRAM offset. Overall, constructing the SRAM address thus requires at least 5 cycles, as realized in Lines 41–45, 49–53, and 57–61.

Read and CHK (Lines 25–64)

The target platform features two dedicated classes of instructions for reading PROM and SRAM. As the destination addresses are determined at runtime, only indirect memory reads are applicable, implying the use of $\text{lpm}$ and $\text{ld}$ with their respective hardwired clock cycles. Since all memory load instructions can only perform a simple verbatim copy of the desired memory word into one of the CPU registers (PM), at least two additional operations are needed to perform the desired $\oplus$ operation of $s_{i,j}$ into the respective positions of $r_i$. Hence, reading and merging a PROM and SRAM block requires at least 4 or 5 instructions, respectively, as realized in Listing 6.1. The PROM reads and checksum merges are performed in Lines 28–30, 33–35 and 38–40, while the SRAM is sampled in Lines 46–48, 54–56 and 62–64.

Loop Control (Lines 78–86)

The loop control block fundamentally requires decrementing the loop counter $i$ and a conditional branch. Decrementing $i$ can be done using a 16-bit subtraction, requiring two clock cycles in total. However, in 255 out of 256 cases we can save one instruction if we split the subtraction into two 8-bit operations, decrementing the higher byte separately and only if decrementing the lower byte resulted in an overflow.

The conditional branches supported by the CPU take a single cycle if the branch is not executed and two cycles otherwise. Hence, we optimize the branching for the most common case that the loop continues to run (condition is not met). Further, the branch

Footnote 9: Unless the memory regions $0x00-0x60$ and $0x100-0x160$ happen to be largely identical, in which case one of them should first be re-initialized with random data or the extra instruction for the carry should be removed.
Listing 6.1: Complete assembly code of the time-critical prover algorithm.

```assembly
1: ; ; Register Layout
4: m28: m29 Y: Loop counter:
5: m30: m31 Temporary (a, and s[1])
6: 7 main_loop: ; Loop from N/3 to 0
8: ; ; GEN: 3 PRNG rounds with byte swapping
9: add r12, r17 ; A := rotr(D, 8)
10: add r13, r27 ; B := rotr(A, 8)
11: add r14, r24 ; C := rotr(B, 8)
12: add r15, r25
13: add r16, r26 ; D := rotr(C, 8)
14: add r17, r22
15: add r18, r23
16: add r19, r4
17: add r20, r19 ; C := rotr(D, 8)
18: add r21, r16
19: add r22, r17
20: add r23, r18
21: add r24, r23 ; D := rotr(C, 8)
22: add r25, r20
23: add r26, r21
24: add r27, r22
25: ; ; Read and merge 3 blocks from PROM and SRAM each
26: mov w Z, r16 ; PROM 1: a_i,1 = r[17,16]
27: andi Zh, 0xF
28: lpm Zh, Zh ; Read from PROM, overwriting Zh
29: xor r4, Zh
30: xor r6, Zh ; Merge twice for better diffusion
31: mov w Z, r18 ; PROM 2: a_i,2 = r[18,18]
32: andi Zh, 0xF
33: lpm Zh, Zh
34: xor r8, Zh
35: xor r10, Zh
36: mov w Z, x20 ; PROM 3: a_i,3 = r[21,20]
37: andi Zh, 0xF
38: lpm Zh, Z
39: xor r8, Zh
40: xor r10, Zh
41: mov Zl, x22 ; SRAM 1: a_i,4 = r[00, x22] + 0x60
42: idi Zh, 0x60 ; Memory region starts at 0x60
43: addi Zl, Zh ; 256 SRAM blocks: 8 random bits
44: clr Zl
45: adcl Zl, Zh ; Recover carry from previous add
46: idi Zh, 0x60
47: xor r1, Zh
48: xor r3, Zh ; Merge twice for better diffusion
49: mov Zl, x23 ; SRAM 2: a_i,5 = r[00, x23] + 0x60
50: idi Zh, 0x60
51: addi Zl, Zh
52: clr Zl
53: adcl Zl, Zh
54: idi Zh, 2
55: xor r5, Zh
56: xor r7, Zh
57: mov Zl, x24 ; SRAM 3: a_i,6 = r[00, x24] + 0x60
58: idi Zh, 0x60
59: addi Zl, Zh
60: clr Zl
61: adcl Zl, Zh
62: idi Zh, 2
63: xor r5, Zh
64: xor r7, Zh
65: ; ; CHK confusion/diffusion
66: add r0, r11 ; H := rotr(T,8)
67: add r1, r8
68: add r2, r9
69: add r3, r10
70: add r4, r3 ; S := rotr(R,8)
71: add r5, r0
72: add r6, r1
73: add r7, r2
74: add r8, r7 ; T := rotr(S,8)
75: add r9, r4
76: add r10, r6
77: add r11, r8
78: ; ; Loop control (require long jump to main_loop)
79: count_l: dec Yl
80: beq count_hl
81: jmp main_loop
82: count_hl:
83: dec Yh
84: beq end
85: jmp main_loop
86: end:
```
Table 6.1: Instructions of CE used in Listing 6.1 and their clock cycles.

**Arithmetic operations:**

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Functionality</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (adc), dec</td>
<td>Add word (with carry), or decrement word</td>
<td>1</td>
</tr>
<tr>
<td>eor, andi, clr</td>
<td>logical $\oplus$, AND or reset a single word</td>
<td>1</td>
</tr>
</tbody>
</table>

**Branching operations:**

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Functionality</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>breq</td>
<td>Branch on zero/equal (else, continue)</td>
<td>1 (2)</td>
</tr>
<tr>
<td>rjmp</td>
<td>Explicit, relative jump (long jump)</td>
<td>2</td>
</tr>
</tbody>
</table>

**Data transfer:**

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Functionality</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldi, ld, lpm</td>
<td>Load from constant, SRAM or PROM</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>mov, movw</td>
<td>Move (copy) word/doubleword</td>
<td>1</td>
</tr>
</tbody>
</table>

instructions can only jump a certain distance from the current position in the program code, which is exceeded in Listing 6.1. Hence, they must be followed by a constant time “long jump” instruction rjmp. This way, decrementing the loop counter $i$ and conditional branching are done in one clock cycle each (except in corner cases), making it the fastest possible loop control block for the considered Computing Engine (CE).

**Pipeline and Memory Bus Saturation**

Instruction reordering can often lead to performance improvements due to better pipeline and address bus saturation, as it is routinely performed by optimizing compilers.

However, we argue that better memory bus saturation cannot be achieved, since the time-critical computations are all performed within the CPU’s registers. The only access to SM that could be accelerated with instruction reordering is the one caused by Read. However, since no memory writes occur, Read is only limited by the speed with which the required pseudo-random memory addresses $a_{i,j}$ become available. In our implementation, we perform these reads as soon as the respective address is calculated, with several additional instructions to compute the respective next address. Hence, no speedup can be expected from reordering memory bus access. Another well-known source of performance optimizations is the saturation of the CPU instruction pipeline. However, the instruction pipeline of the targeted low-cost systems, and in particular our prototype platform, has only a single step, so that pipeline optimization has no effect on our platform.

Overall, we thus argue that no performance can be gained through reordering or replacement of instructions. That is, there exists no sequence of instructions which succeeds computing the checksum with significant probability in less time.

**6.5.5 Atomicity of GEN, Read, and CHK**

A drawback in the way our assumptions are currently formulated is that we require the basic building blocks of the attestation algorithm, Gen, Read, and CHK, to be atomic. In particular, we assume that a partial execution of these components does not yield any information on their outputs. On the other hand these components cannot be atomic
since $g$ and $r$ must be much larger than the word size of our target platform to assure the cryptographic security of our scheme, and thus can be computed only iteratively. Hence, the adversary could simply skip a part of these composed functions to save execution time at the cost of computation inaccuracies.

However, from a practical perspective the problem is more contrived. If we consider for instance the upd operation in Lines 8–24 and 65–77, it is easy to see that removing even a single instruction implies an average uncertainty in 8 bits of the computed values, affecting all subsequent rounds unless some effort is invested later to correct the error. The same argument holds for the instructions in Lines 25–64 and to the loop in Lines 78–86.

For example, by removing any of the carry bit additions in Lines 45, 53 or 61, the adversary has a chance of 62% to save an instruction without reading and merging a wrong memory block $s_{i,j}$. However, the adversary cannot determine when the bit error will occur without again inserting extra instructions. Hence, to save any reasonable amount of time, there is a high chance that wrong memory blocks are merged in some checksum iterations, creating a bit error that will be amplified by the avalanche property of $\text{CHK}$ to quickly and irrevocably manipulate the checksum value. Also, the cost of a conditional branch that triggers depending on the current carry bit state requires at least one extra clock cycle, eliminating any potential gain of removing the carry addition. Finally, any adversary that manipulates the checksum code must also intercept the sampling of program memory such that the manipulation itself is not detected by the checksum.

Overall, it can be seen that even though the individual components are not atomic, any conceivable manipulation results in wrongly calculated intermediate results which, amplified by the checksum’s avalanche property, have an overwhelming chance to influence all subsequent computations and are costly to compensate.

Remark 5: Atomic Variations of $\text{Attest}$. Note that a target platform with 32 or 64 bit CPU may still be able to realize $\text{Gen}$, $\text{Read}$, and $\text{CHK}$ as atomic, e.g., by using complex native instructions like AES-NI or T-functions [179] as $\text{Upd}$ and highly simplified versions of $\text{Sel}$ and $\text{Read}$. $\text{Gen}$ can then be realized as a variation of $\text{Upd}$, as already done in our proposed instantiation above.

6.6 Discussion

We have shown that purely software-based attestation can be secure in a cryptographic sense, despite the fact that it uses no cryptographic secrets and often also no cryptographic primitives. In particular, we have shown that a timed self-checking algorithm can feature a fairly simple design and we provide an estimate for the maximum adversary advantage. In the following we discuss how these results relate to prior work and in particular, where and why we diverged from it.

Incompressibility of Secondary Memory

Our analysis exposes the implicit assumption of previous works, that the state $S$ of the prover $\mathcal{P}$ must have sufficient entropy such that guessing (instead of reading) memory words of the Secondary Memory (SM) is inefficient (Assumption 2). In particular, while
some previous works consider the replication, compression, and de-compression of memory to improve or to attack software attestation [335, 74], our approach comprehensively summarizes the adversary's success probability as a function of the probability of guessing the content of the prover's state at a particular address.

Fortunately, due to the large number of pseudo-random memory samples that are usually taken during a single attestation protocol-run, typical program code contains sufficient entropy to make consistent guessing very hard. However, as firmware images often also contain constant data (all zeros or all ones) in unused memory regions, a case-by-case analysis is required. For instance, 30 out of 233 firmware images hosted in a typical Debian/Linux installation contain more than 50% zero bytes, which may facilitate an attack if used with software attestation.

Efficient Memory Sampling

To determine the desired number of memory samples $N$ used by the attestation algorithm Attest, previous works employed the coupon collectors problem. In particular, it was assumed that, to assure that every memory block of the prover's state $S$ is measured at least once with high probability, $N$ must be $\geq s \cdot \log(s)$ [275, 192]. However, as it is hard for the adversary to determine which of the state entries in $S$ are not measured, a significantly smaller amount of samples may be sufficient. On the other hand, several other parameters influence the overall adversary advantage, including the collision probability of the checksum function, the fraction of potentially modified memory blocks and the entropy of the good state.

Based on our model, one can now give a more accurate estimate of the worst-case adversary advantage given a particular set of parameters and one can choose $N$ appropriately. Moreover, our instantiation of CHK provides a general approach to sample multiple memory blocks in a single checksum round, reducing the computational overhead and allowing to exploit hardware features such as read-ahead without loss of security.

Checksum Function Design

Previous works often refrain from using strong collision-resilient primitives, such as cryptographic hash functions, due to their relatively high complexity and computation overhead. Instead, several constructions followed the idea of a "strictly ordered" checksum function design, a notion first introduced in Pioneer [273]. The motivation is that the adversary should not be able to re-order operations for possible code optimization and that swapped memory blocks are detected with high probability.

However, it was unclear whether such constructions are sufficient to create a $2^{nd}$ preimage resistant checksum, and to which degree a particular algorithm can be "strictly ordered". In particular, at least two prior works following this approach are known to suffer from $2^{nd}$ preimage attacks (cf. Section 6.5.3 and [74]). Based on our analysis, we separate the informal notion of a "strictly ordered" checksum function into two specific requirements (Assumptions 4 and 5), and demonstrate their efficient realization in Section 6.5. In particular, we introduce a heuristic test based on the well-known avalanche test to evaluate and compare the security of our and several previous checksum constructions.
6.7. RELATED WORK

Memory Copy Attack

The memory copy attack is a runtime attack where the adversary executes a modified attestation algorithm that measures a copy of the original (unmodified) code and data. This way, the execution can divert or return to malicious code after the time-critical main loop of the Attest algorithm has computed the correct checksum. Hence, the device stores the expected software state but does not execute it in the desired way.

The attack is prevented by assuring that the measured code is the currently executed code. Previous works used the CPU's Instruction Pointer to assure that the executed code is part of the measured memory area [272, 271], or employed randomized jumps to prevent manipulations of the program counter [273, 192]. However, as we strive for a mostly platform-independent analysis and design of software attestation, we instead followed the approach of measuring the whole program memory, including any currently executed code. This way, it is impossible for the adversary to hide a modified copy of the attestation algorithm without significant overhead for inspecting, redirecting and extracting current memory accesses. In particular, we showed how to systematically exploit the memory hierarchy to partition the memory into PM and SM such that only a reduced amount of SM must be measured (similar as in [121, 318]) and model this approach based on the implicit delay induced by Read.

6.7 Related Work

Several works considered the design and extension of software attestation. Existing literature focuses on the design of checksum constructions for different platform architectures and countering platform-specific attacks [275, 273, 113, 272, 192]. Several works consider the strengthening of self-checksumming code against malicious modifications by either limiting the memory available to the prover during attestation [121, 318] or by using self-modifying and/or obfuscated attestation algorithms [277, 129]. Since software attestation does not rely on any secrets and is thus fundamentally unable to authenticate the prover to the verifier, multiple works also consider how to combine software attestation with hardware trust anchors such as TPMs and SIM-cards [268, 185, 160], or intrinsic hardware characteristics such as code execution side-effects [171, 275, 278] or Physically Unclonable Functions [16]. Interestingly, most proposed implementations employ hash functions and PRNGs that are not cryptographically secure. Further, works that used cryptographically secure algorithms did not consider whether these components maintain their security properties in the "key-less" software attestation scenario, where the underlying secrets such as PRNG states and the intermediate values of hash functions are known to the adversary. In this respect, our formal analysis and the subsequent systematic design of a practical software attestation checksum provides a first step towards a deeper and more comprehensive understanding of software attestation. However, due to the large variety of existing schemes and since we aimed for a largely platform-independent framework, we do not currently model approaches using self-modifying code, hardware trust anchors, or execution side-effects to prevent memory copy attacks (cf. Section 6.6).

Similar to software attestation, proofs of work challenge the prover with computationally expensive or memory-bound tasks [95, 96]. However, while the goal of proofs of
work is to mitigate Denial-of-Service and Spam by imposing artificial load on the service requester, the major design goal of software attestation is secure self-measurement using a time-optimal blind 2<sup>nd</sup> preimage resistant checksum. Hence, algorithms for proofs of work tend to be less efficient and not designed for the complex security constraints of software attestation.

A number of works propose lightweight attestation mechanisms based on low-cost hardware modifications. [237] present a local attestation scheme based on ROM and secure proofs of erasure. Assuming a secure key storage and assured memory clearing upon platform reset, SMART [88] shows that remote attestation and Dynamic Root of Trust for Measurement (DRTM) are achievable using only a few specialized memory access control rules (see also Chapter 9).

### 6.8 Summary

We presented the first formal security framework for software attestation, proved the security of a generic software attestation protocol and derived the adversary advantage as a function of various system and design parameters. By formalizing the assumptions of software attestation and arguing in detail how these can be fulfilled, we provide a fundamental first step in the systematic design of software attestation schemes. Our analysis shows that software attestation can be secure, although it uses PRNGs and non-cryptographic hash functions in unexpected ways. In particular, we find that previously only informally defined algorithmic security requirements can probably be much reduced.

For future work, it would be interesting to investigate if the pseudo-randomness requirement on $Gen$ or the required atomicity of $Gen$ and $CHK$ can be relaxed further. Similarly, the proposed blind 2<sup>nd</sup> preimage resilience and avalanche test may be extensible towards a concrete provable construction of $CHK$. Other important problems are the design of optimal checksum code and the extension of our generic scheme to consider the integration of hardware trust anchors.

A fundamental problem for the deployment of software-based attestation in many low-cost/embedded systems is that it is actually rather resource-intensive. This is because the checksum function is designed to completely exhaust the computational capacity of the prover platform and communication jitter may additionally require the use of rather large values of $N$.

Another unusual and much discussed problem is the lack of prover and verifier authentication, as well as the assumption that no third party can collude with the prover during the attestation to assist it in computing the correct checksum result. Even in the often proposed “local” attestation scenarios, it is often conceivable that the hardware may have been manipulated or exchanged without knowledge of the verifier. We discuss the latter two problems in more detail in the following Chapter 7, and also propose a possible solution by combining software-based attestation with Physically Unclonable Functions (PUFs).
Chapter 7

PUF-based Remote Attestation

Software-based attestation [147, 274] is an interesting new concept to establish trust in low-cost, resource-constrained devices, as it does not assume hardware or software components that are trusted by the verifier. Instead, software attestation exploits the computational limits of the prover to ensure that only a specific algorithm can be executed within a certain time frame. Our analysis of software attestation in Chapter 6 shows that it is fundamentally sound, although several questions regarding its fundamental cryptographic requirements remain open.

However, software attestation also suffers from some major system-level drawbacks that make it unsuitable for many practical scenarios. In particular, software attestation is unable to identify the prover itself and instead assumes an out-of-band authentication channel such as visual inspection by the verifier. Further, it assumes that the verifier has detailed knowledge of the prover’s computational capabilities. The approach is thus generally vulnerable to impersonation and relay attacks, such as the partial outsourcing of the computation to colluding devices (cf. [275, 273]). Additionally, software attestation is highly vulnerable to hardware attacks, such as simple CPU overclocking, since a faster prover may be able to forge the attestation checksum.

To overcome these problems, prior works have proposed to link the checksum computation to the prover platform, exploiting hardware-specific side-effects [171] or again assuming trusted co-processors such as Trusted Platform Modules (TPMs) [268, 185] or SIM-card [160]. However, the computational side-effects are insufficient to achieve a strong binding to individual platforms [278, 275], and the inclusion of trusted co-processors seems to contradict the idea of software-based attestation.

In this chapter, we propose the use of Physically Unclonable Functions (PUFs) as a trust anchor in software-based attestation. PUFs exploit unique, random variances in the manufacturing process of a device to produce device-specific, low-cost hardware identifiers that are hard to clone and manipulate. As such they are fundamentally less complex and costly than secure co-processors, and represent highly interesting emerging technology for the design of low-cost cryptographic protocols that also provide a certain tamper evidence.

After introducing PUFs in Section 7.1, we present the high-level PUF-based attestation scheme in Section 7.2. In contrast to purely software-based attestation, our scheme (1) is secure against collusions of malicious provers, (2) allows for the authentication and attestation of remote provers, and (3) enables the detection of certain hardware attacks on
CHAPTER 7. PUF-BASED REMOTE ATTESTATION

the prover. We consider possible instantiations of our scheme in Section 7.3 and discuss different solutions for the efficient and practical verification of PUFs by the verifier in Section 7.4. The security of our approach is analyzed in Section 7.5. Our approach can be viewed as generalization of [171, 192] and allows the deployment of any type of trust anchor that can be abstracted as a Pseudo-Random Function (PRF).

7.1 Physically Unclonable Functions (PUFs)

A Physically Unclonable Function (PUF) is a noisy function that is embedded into a physical object, e.g., an integrated circuit [234]. Today, there are already several PUF-based security products aimed for the market, e.g., PUF-enabled RFID chips and proposals for IP-protection and anti-counterfeiting solutions [322, 155]. When queried with a challenge \( x \), a PUF generates a response \( y \leftarrow \text{PUF}(x) \) that depends on both \( x \) and the unique device-specific intrinsic physical properties of the object containing PUF. Since PUFs are subject to noise (e.g., thermal noise), they typically return slightly different responses when queried with the same challenge multiple times. However, these output variations can be eliminated by using fuzzy extractors [92], which can be efficiently implemented on resource-constrained devices [312]. Hence, PUFs can be used as deterministic functions.

Based on [38, 260], we consider PUFs that have the following properties, where \( \text{PUF} \) and \( \text{PUF}' \) are two different PUFs:

**Robustness:** When queried with the same challenge \( x \), PUF always returns the same response \( y \).

**Independence:** When queried with the same challenge \( x \), PUF and \( \text{PUF}' \) return different responses \( y \) and \( y' \).

**Pseudo-randomness:** It is infeasible to distinguish a PUF from a pseudo-random function.

**Tamper-evidence:** Any attempt to physically access the object containing PUF irreversibly changes PUF, i.e., PUF cannot be evaluated anymore but is turned into a random PUF \( \neq \text{PUF} \).

Independence and pseudo-randomness imply that \( \mathcal{A} \) cannot predict PUF responses to unknown challenges or decide with probability larger than 50% if a given response was created by PUF or PUF'. This also implies that \( \mathcal{A} \) cannot simulate a PUF based on partially known challenge/response behavior. Moreover, tamper-evidence ensures that \( \mathcal{A} \) cannot obtain any information on the PUF by physical means, e.g., hardware attacks. Hence, \( \mathcal{A} \) cannot simulate or clone a PUF.

7.2 PUF-based Attestation

Our PUF-based attestation scheme extends existing software attestation protocols. A software attestation protocol is a two-party protocol between a prover \( \mathcal{P} \) and a verifier \( \mathcal{V} \),
7.2. PUF-BASED ATTESTATION

where \( V \) should be convinced that \( P \) is in a trusted software state \( S \). Typically, \( P \) is an embedded device with constrained computing capabilities (e.g., a sensor node), whereas \( V \) is a more powerful computing device (e.g., a base station). On a high level, all known software attestation protocols exploit the computational limits of \( P \) to assure that nothing else than a specific trusted algorithm can be executed within a specific time frame.

In contrast to existing software attestation schemes, our solution assures the verifier \( V \) that the attestation result has actually been computed by the original hardware of a specific prover \( P \). We propose to use a hardware checksum based on PUFs to include device-specific properties of \( P \)'s hardware into the attestation protocol. Our design exploits the limited throughput of external interfaces to prevent an adversary from outsourcing the computation of the software checksum to a more powerful computing device.

7.2.1 Trust Model and Assumptions

The adversary \( A \) controls the communication between the verifier \( V \) and the prover \( P \), i.e., \( A \) can eavesdrop, manipulate, reroute, and delete all messages sent by \( V \) and \( P \). Moreover, \( A \) knows all algorithms executed by \( P \) and can install malicious software on \( P \). However, due to the unclonability of the PUF, \( A \) cannot simulate the hardware checksum, while the tamper-evidence of the PUF ensures that \( A \) cannot physically access or manipulate the internal interfaces between CPU, memory, and PUF of \( P \). Further, we assume that external interfaces of \( P \) are significantly slower than the internal interface that is used by the CPU to access the PUF. All provers \( P \) are initialized in a secure environment before deployment. The verifier \( V \) is trusted to behave according to the protocol. Moreover, \( V \) can emulate any algorithm that can be executed by \( P \) in real time and maintains a database \( D \) containing the identity \( I \) and the exact hard- and software configuration of each \( P \).

7.2.2 Extension of Software Attestation Scheme

Figure 7.1 shows the proposed PUF-based attestation protocol, consisting of a generalized software-attestation protocol with additional inclusion of a device-characteristic hardware checksum\(^1\) function \( \text{HwCHK()} \) at the prover \( P \) and \( \text{EmulateHwChk()} \) at the verifier \( V \). By careful integration of this hardware checksum into the software attestation algorithm, we bind the software attestation to the respective hardware platform, enabling remote attestation.

The main protocol is the generalization of a typical software attestation protocol: The verifier \( V \) starts by sending a random challenge \( c \) to the prover \( P \) and then measures the time \( P \) takes to reply with the checksum \( r_N \) computed over its current software state \( S \) (e.g., its program memory). In detail, on receipt of \( c \), \( P \) sets up the initial algorithm state \( (g_0, r_0, x_0) \leftarrow c \) as required by the underlying software attestation scheme. \( P \) then iteratively computes \( r_N \) by taking \( N \) random measurement samples out of \( S \). Specifically, in each iteration \( i \) of the checksum computation, \( P \) invokes three procedures: \( \text{Gen()} \), \( \text{CHK()} \), and \( \text{HwCHK()} \). \( \text{Gen}(g_{i-1}, y_{i-1}) \) is used to generate an updated internal state \( g_i \) and a memory

\(^1\)For the purpose of this work, we consider \( \text{HwCHK()} \) to be implemented as a PUF to gain tamper-evidence. However, simpler implementations are possible, e.g., an HMAC with a hard-wired key.


<table>
<thead>
<tr>
<th><strong>Verifier V</strong></th>
<th><strong>Prover P</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Stores D = {..., (I, S₀, C₀, δ₀), ...}</td>
<td>Stores (I, S)</td>
</tr>
<tr>
<td>Choose random challenge c</td>
<td>(g₀, r₀, y₀) ← c</td>
</tr>
<tr>
<td>Save current time t'</td>
<td>for i = 1 to N do</td>
</tr>
<tr>
<td>(gᵢ, rᵢ, yᵢ) ← c</td>
<td>yᵢ ← HwChk(rᵢ₋₁)</td>
</tr>
<tr>
<td>for i = 1 to N do</td>
<td>(aᵢ, gᵢ) ← Gen(gᵢ₋₁, yᵢ₋₁)</td>
</tr>
<tr>
<td>yᵢ ← EmulateHwChk(Cᵢ, rᵢ₋₁)</td>
<td>rᵢ ← CHK(rᵢ₋₁, sᵢ)</td>
</tr>
<tr>
<td>(aᵢ, gᵢ) ← Gen(gᵢ₋₁, yᵢ₋₁)</td>
<td>end</td>
</tr>
<tr>
<td>rᵢ ← CHK(rᵢ₋₁, sᵢ)</td>
<td>end</td>
</tr>
<tr>
<td>Save current time t'</td>
<td>rₙ</td>
</tr>
<tr>
<td>if (t₂ - t₁) ≤ δ₁ and rₙ = rₙ then accept P</td>
<td>else reject P</td>
</tr>
</tbody>
</table>

Figure 7.1: Remote attestation based on physical functions.

address aᵢ, which determines the next memory block sᵢ ← Read(S, aᵢ) to be included into the software checksum as rᵢ ← CHK(rᵢ₋₁, sᵢ). Note that CHK() is the same function as in plain software attestation (cf. Chapter 6), while we require only a minor modification of Gen() to include the additional hardware checksum outputs yᵢ₋₁. Typically, modern software attestation schemes implement Gen() as a Pseudo-Random Number Generator (PRNG) to prevent efficient pre-computation or memory mappings attacks. However, neither the PRNG nor the CHK() are usually required to be cryptographically strong [275]. In any case, since yᵢ₋₁ can be considered random, it is usually straightforward to integrate yᵢ₋₁ into Gen() as an extension of its internal state, e.g., gᵢ ← yᵢ + gᵢ.

In contrast to plain software attestation, our attestation scheme integrates a hardware checksum HwCHK() into each iteration i, yielding the previously mentioned additional input yᵢ ← HwCHK(rᵢ₋₁) to the Gen() procedure. As a result, every iteration of the software checksum additionally depends on the result of the device-specific hardware checksum, thus binding the attestation response rₙ to the prover’s hardware. Similarly, each iteration of HwCHK() depends on the previous intermediate software checksum rᵢ₋₁, such that HwCHK() cannot be executed independently of CHK(). Note that HwCHK() can still be executed in parallel to CHK() but must be tightly synchronized: rᵢ₋₁ is already available at the start of round i and the output yᵢ is only required in the next round i + 1.

After every memory block sᵢ has been included into the checksum at least once, P sends rₙ to V. While waiting for the response of P, V can compute a reference checksum rₙ by emulating the computation of P using the known trusted software state Sᵢ recorded in database D and emulate HwCHK() using EmulateHwChk() with some verification data Cᵢ, which is secret information only available to V. V accepts only if (1) P replied within a certain time frame δᵢ and (2) rₙ matches rₙ. The first check ensures that P computed rₙ in about the same time δᵢ an honest device would have needed and has not performed additional computations, e.g., to hide the presence of malware. The second check verifies whether the software state S measured by P corresponds to the known trusted software state Sᵢ. If either of these checks fails, P is assumed to be in an unknown software state.
and is rejected.

Note that the verification of the PUF-based hardware checksum by $V$ is not straightforward: $V$ must be able to predict the outputs of the PUF, while this must be infeasible for $A$. This is further complicated by the large amount of hardware checksum responses required by our construction and the closely parallelized execution of software and hardware checksum. Hence, the integration of PUFs into software attestation requires careful consideration.

### 7.2.3 Security Objectives

In contrast to existing software attestation schemes, our PUF-based attestation scheme additionally achieves the following security goals:

**Correctness:** Provers in a known trusted state must always be accepted by the verifier.

**Unforgeability:** A prover that is in an unknown state must be rejected by the verifier.

**Prover isolation:** A prover colluding with other (malicious and/or more powerful) devices to forge the attestation must be rejected by the verifier.

**Prover authentication:** Any other device pretending to be a particular known prover must be rejected by the verifier.

**Tamper-evidence:** A prover that is not in its original hardware state must be rejected by the verifier.

### 7.3Instantiation

In this section, we show how existing software attestation schemes can be used to instantiate software checksum $\text{CHK}()$ and the memory address generator $\text{Gen}()$ with only minor modifications. Moreover, we discuss different instantiations of the hardware checksum $\text{HwCHK}()$ and, in particular, the corresponding secret verification data $C_i$ and $\text{EmulateHwChk}()$ algorithm.

#### 7.3.1 Memory Address Generation and Software Checksum

The memory address generator $\text{Gen}()$ and the software checksum $\text{CHK}()$ components of our PUF-based attestation scheme can be instantiated based on any existing software-based attestation scheme (e.g., [272, 335, 81]) with only minor modifications to the underlying $\text{Gen}()$ algorithm to integrate the hardware checksum $\text{HwCHK}()$. In all modern software attestation designs, $\text{Gen}()$ is implemented as a PRNG with internal state $g_i$ that is used to generate pseudo-random memory addresses $a_i$. We can thus integrate the output $y_{i-1}$ of $\text{HwCHK}()$ simply by merging it with the current state $g_i$ in each iteration. Due to the unpredictability property of the PUF (Section 7.1), this is equivalent to a state expansion of the PRNG to encompass the PUF challenge-response behaviour, effectively preventing the PRNG from repeating its sequence.
Algorithm 2: Gen() of SCUBA (modified).

Input:
- $g$: current PRNG state
- $a$: current memory address
- $y$: response of last hardware checksum query
- $\text{code}_{\text{start}}$, $\text{MASK}$: memory address range to be measured

Output:
- $g$: updated PRNG state
- $a$: next memory address

SCUBA-GenMemAddr($g, a, y$)

begin
  // T-function, updates $g$ where $0 \leq g < 2^{16}$
  $g \leftarrow g + (g^2 \lor 5) \oplus y \mod 2^{16}$
  // Compute random memory address using $g$
  $a = ((a \oplus g) \land \text{MASK}) + \text{code}_{\text{start}}$
  return $g, a$
end

We use Secure Code Update By Attestation (SCUBA) [272] as an example to demonstrate the integration of existing software attestation schemes into our framework. We selected SCUBA since it targets resource-constrained devices and is presented with sufficiently detailed pseudo-code and experimental results.

Algorithm 2 depicts the (modified) Gen() algorithm of SCUBA, which implements a PRNG based on a T-function [179] that iterates over all possible PRNG states $g$. The output of the PRNG is used to generate a random memory address $a$ in the range specified by the verifier. Algorithm 3 implements CHK(), which is iteratively used to update the intermediate checksum value $r$. This update is done by periodically iterating through the ten 16 bit parts of the 160 bits of $\sigma$. We modify Algorithm 2 such that the state update function of the PRNG (i.e., the T-function) integrates the output $y$ of HwCHK(), which makes the new PRNG state $g$ dependent on $y$. Since the state update function iterates through all possible values of $g$, our modification introduces random jumps into the pseudo-random sequence, which does not weaken the security of the PRNG. Note that similar modifications are possible with any other PRNG.

Based on Algorithm 2 and 3, the complete PUF-based attestation scheme is an instantiation of our high-level protocol design depicted in Figure 7.1.

7.3.2 Hardware Checksum

We present two alternative instantiations of the hardware checksum HwCHK() based on emulatable and non-emulatable PUFs. In general, emulatable PUFs yield more efficient protocols. However, since PUFs are not expected to be emulatable by design (Section 7.1), we focus on solutions for different approaches based on non-emulatable PUFs.
Algorithm 3: CHK() of SCUBA.

Input:
- \( r \): current checksum value
- \( g \): current PRNG state
- \( a \): address of current memory block
- \( i \): loop counter (i.e., \( i \)-th iteration of CHK())

Output:
- \( r \): updated checksum value

Data:
- \( j \): determines the part of the checksum to be modified
- \( SR \): current CPU status (flags) register value
- \( PC \): current CPU program counter value

```plaintext
SCUBA-SwSum(r, g, a, i)
begin
    // Calculate checksum at index \( j \)
    // Note: \( S[a] \) denotes block \( s \) of \( S \) at address \( a \)
    \( r_j \leftarrow r_j + PC \oplus S[a] + i \oplus r_{j-1} + g \oplus a + r_{j-2} \oplus SR \)
    \( r_j \leftarrow \text{rotate-left}(r_j) \)
    // Update checksum index (10 \cdot 16 = 160 bit)
    \( j \leftarrow (j + 1) \mod 10 \)
end
return \( r \)
```

Emulatable PUFs

One approach to implement HwCHK() are emulatable PUFs, which allow the manufacturer of the PUF to set up a mathematical model that enables the prediction of PUF responses to unknown challenges [232, 254]. Typically, the creation of this model requires extended access to the PUF hardware, which is only available during the manufacturing process of the PUF and permanently disabled before deployment [232].

More detailed, during the production of the hardware of prover \( P \), the trusted hardware manufacturer sets up a secret mathematical model \( C_I \) of PUF(). Before deployment of \( P \), the interface for modeling the PUF() is then disabled such that any attempt to reactivate it leads to an irreversible change of PUF(). During deployment of \( P \), \( C_I \) and an algorithm EmulateHwChk() for emulating HwCHK() is given to the verifier \( V \). In the attestation protocol, \( P \) computes HwCHK() = PUF(), whereas \( V \) emulates HwCHK() = EmulateHwChk(C_I, ·).

In practice, emulatable PUFs can be realized by most delay-based PUFs (e.g., Arbiter PUFs [191, 126] and Ring Oscillator PUFs [125]), which allow for creating precise mathematical models based on machine learning techniques [255]. However, the security properties of practical instantiations of emulatable PUFs still need further evaluation. Hence, in the following section, we present different solutions based on non-emulatable PUFs.
Non-emulatable PUFs

For non-emulatable PUFs, the verifier $V$ typically maintains a secret database $D$ of PUF challenges and responses, called Challenge Response Pair (CRP) database. Note that our attestation scheme requires PUFs that ideally have an exponentially large CRP space, such that an adversary $A$ with direct access to the PUF cannot create a complete CRP database and then emulate the PUF. However, this means that $V$ can store only a subset of the CRP space. We thus have to deterministically limit the CRP subspace used during attestation without allowing $A$ to exploit this to simulate the PUF in future attestation runs. In the following, we describe two different approaches for how to instantiate $\text{HwCHK()}$ with non-emulatable PUFs.

Commitment to Procedure. One approach is creating a database $D$ of attestation challenge messages $c$ and the corresponding checksums $r_N$ in a secure environment before the prover $P$ is deployed. In the attestation protocol, the verifier $V$ can then use $D$ to obtain the reference checksum $r_N$ instead of emulating the PUF.

Specifically, before deployment, $V$ runs the attestation protocol several times with $P$ in a secure environment. For each protocol run, $V$ records in $D$ the attestation challenge $c$ sent to $P$ and the corresponding checksum $r'_N$ returned by $P$. When running the attestation protocol after deployment, $V$ chooses a random set $(I, c, r'_N, \delta_I) \in D$ and sends $c$ to $P$, which then computes $r_N$ using $\text{HwCHK()}$. $V$ accepts only if $P$ replied with $r_N = r'_N$ in time $\delta_I$.

The solution allows for very efficient verification of $r_N$ by $V$, however, the number of attestation protocol runs of each $P$ is limited by the size of $D$. Moreover, this approach does not allow updating the software state of $P$ after deployment, e.g., to fix bugs that allow runtime compromise.

Commitment to Challenge. Since updates to the software of the prover $P$ are usually developed after deployment of $P$, the software state $S$ and thus the inputs to $\text{HwCHK()}$ are not known before deployment of $P$ and the final checksum value $r_N$ cannot be computed in advance.

Our solution to this problem is to reduce the amount of possible inputs $x_i$ to $\text{HwCHK()}$ generated by the intermediate checksum results $r_{i-1}$, such that it becomes feasible to create a CRP database for node that is independent of $r_i$, and thus $S$. To prevent the adversary from exploiting this to simulate the attestation procedure, we use a random offset $q$ to determine this reduced CRP space within the overall CRP space of $\text{HwCHK()}$, such that the adversary cannot generate the required CRPs before the actual attestation protocol starts. The offset $q$ is sent from the verifier $V$ to $P$ together with a random attestation challenge $c$ in the first message of the attestation protocol (Figure 7.1).

Specifically, we chose $f(\cdot)$ to be a function that maps intermediate checksum results $r_{i-1}$ to bit strings of length $n$ and derive the input to $\text{HwCHK()}$ as $x_i \leftarrow q \parallel f(r_{i-1})$. Before deployment, the verifier $V$ evaluates $y_j \leftarrow \text{HwCHK}(q \parallel j)$ for all $j \in \{0, \ldots, 2^n - 1\}$, and records $(q, y_0, \ldots, y_{2^n-1})$ as $C_{I,q}$ in $D$ for a number of randomly chosen offsets $q$.

After deployment, $V$ chooses a random entry $(I, S_I, C_{I,q}, \delta_I) \in D$ and a random nonce $c$ to start the attestation. The prover $P$ then computes the checksum $r_N$ using
Algorithm 4: Low-speed hardware checksum $\text{HwCHK}'()$.

**Input:**
- $x$: challenge to $\text{HwCHK}'()$
- $v$: number of iterations

**Output:** $y$: response of $\text{HwCHK}'()$

**Data:** $u$: loop counter

$\text{HwSum}'(x, v)$

begin
  $y \leftarrow 0$
  for $u = 0$ to $v$ do
    $y \leftarrow y \oplus \text{HwCHK}(u || y \oplus x)$
  end
  return $y$
end

$\text{HwCHK}(q || f(r_{i-1}))$. While waiting for the response of $P$, $V$ computes the reference checksum $r'_N$ using $\text{EmulateHwChk}(C_{l,q}, q || f(r'_{i-1}))$ and the current reference software state $S_I$. $V$ accepts only if $P$ replied with $r_N = r'_N$ in time $\delta_I$.

In this approach, the number of attestations are limited by the amount of random offsets $q$ for which a CRP subspace has been generated in advance and by the storage available at the verifier $V$. The offsets cannot be re-used since they cannot be encrypted and would enable the adversary to pre-compute or even replay future attestation protocol runs.

### 7.4 Practical Considerations

In the following, we discuss some interesting variations of our attestation scheme to support its practical integration.

#### 7.4.1 Checksum Synchronization

Our protocol assumes a strong inter-dependency and highly concurrent execution of the hardware and software checksum functions at the prover. However, in practice, the software and hardware checksum functions are limited by additional constraints such as cost, power consumption, and availability.\(^3\) In the following we thus present two alternatives for a more abstract hardware checksum function $y_i \leftarrow \text{HwCHK}'(x_i)$ as a wrapper to the actual hardware checksum $\text{HwCHK}()$, which adapts the average access time of the hardware checksum to that of the software checksum and thus increases their inter-dependency.

\(^2\)Such encryption would require a secret key that must be protected against SW attacks. However, by assumption the software of $P$ is compromised and no trusted key storage exists.

\(^3\)In particular, current PUF constructions are not designed or evaluated for high access speeds. We are currently investigating suitable PUF prototypes for the practical evaluation of our protocol.
Algorithm 5: High-speed hardware checksum $HwCHK'()$.

**Input:** $x$: challenge to $HwCHK'()$

**Output:** $y$: response of $HwCHK'()$

**Data:**
- $v$: PRF re-seeding period, fixed hardware parameter
- $u$: loop counter, initialized as 0
- $seed$: temporary PRF seed

$HwSum'(x)$

begin
  if $u = 0 \mod v$ then
    $seed \leftarrow HwCHK(x)$
  end
  $y \leftarrow PRF(seed, x)$
  $u \leftarrow u + 1$
  return $y$
end

In case the real hardware checksum $HwCHK()$ is faster than $CHK()$ and $Gen()$, we use a logical hardware function $HwCHK'()$ that simply issues $v > 1$ queries to the underlying $HwCHK()$ in a cipher-feedback mode, as illustrated in Algorithm 4. Note that Algorithm 4 enforces multiple subsequent queries to $HwCHK()$ to derive the final output of $HwCHK'()$ and may thus also be implemented in software. We include a counter $u$ into the input of $HwCHK()$ for additional diffusion of its previous outputs $y$. However, several other diffusion functions are possible.

On the other hand, if the software checksum is faster than the hardware checksum, we require a construction where the actual $HwCHK()$ is queried less often than the software checksum. Unfortunately, this reduces the inter-dependency of the two algorithms and may allow an adversary to violate, e.g., prover isolation (Section 7.2.3). To mitigate this problem, we propose the hardware checksum construction depicted in Algorithm 5, where a fast $HwCHK'()$ is implemented in hardware, using a fast PRF that is continuously re-seeded by $HwCHK()$. If the seeding period is sufficiently low, this construction also remains secure against side-channel attacks since each PRF seed is only valid for a few microseconds. To achieve tamper-resistance, the implementation of $HwCHK'()$ must be protected as one of the security-critical device components, e.g., by physically shielding it with a coating PUF [313, 324].

Note that multiple $HwCHK()$ functions may be integrated into the device, e.g., to provide comprehensive tamper-evidence for all hardware components through multiple PUFs. Moreover, they can be incorporated into Algorithm 5 to lower the frequency of accessing the individual $HwCHK()$ functions. Hereby, a global counter iterates over the number of available $HwCHK()$ functions and accesses them successively.

For example, the authors of [272] evaluate the SCUBA protocol for $N = 40,000$ and measure a legitimate attestation time frame of about 2.87 seconds. When instantiating our protocol based on SCUBA, the PUF should operate at $40,000/2.87 \approx 14$ kHz. Assuming
that the PUF cannot operate at this frequency but only reaches, e.g., 200 Hz \[84\], and assuming that two PUFs are available per device \( P \), we can use Algorithm 5 and access the two PUFs in turns, resulting in an optimal PRF re-seeding period of about \( v = 14 \text{ KHz} / (2 \cdot 200 \text{ Hz}) = 35 \) or 2.5 milliseconds.

Finally, a special case of Algorithm 5 is where the PRF is seeded only once per attestation and the initial \( \text{HwCHK}() \) challenge is a shared secret between prover and verifier (i.e., is stored in the prover's hardware to resist software compromise), which can be protected against hardware attacks using key storage based on PUFs \[325, 195\].

### 7.4.2 Key Establishment and CRP Generation

We propose a method to reduce the storage requirements at the verifier \( V \) and to allow a theoretically unlimited number of attestation protocols runs, by generating additional CRP subspaces on demand, once an attestation succeeded.

Specifically, \( V \) and \( P \) can establish a mutually authenticated and confidential channel after successful attestation to exchange additional CRPs for future attestation runs. For this purpose, \( r_N \) is treated as a common shared authentication secret and the proof of knowledge of this secret (the last message of the attestation protocol in Figure 7.1) is replaced with an authenticated key exchange.

However, contrary to the original proposal in \[16\] one cannot use \( r_N \) directly as a session key. As pointed out to us by Dries Schellekens, an adversary can recover \( r_N \) by recording the protocol messages exchanged between \( P \) and \( V \), then compromising the software of \( P \), and recompute \( r_N \) by replaying the verifier’s challenge \( c \) and simulating the attestation using \( \text{HwCHK}() \) of \( P \) as an oracle. Knowing \( r_N \), the adversary can decrypt the required CRPs and future challenges \( c \) exchanged with the on-demand CRP generation protocol of \[16\]. The gained a priori knowledge on future CRPs compromises all future (remote) attestations.

To mitigate this attack, \( r_N \) should be interpreted as an authentication secret for an authenticated key exchange that provides forward secrecy, such as authenticated Diffie-Hellman. Although this makes the solution less lightweight, the whole process of on-demand CRP generation can be rather resource intensive due to the exchange of the several thousand CRPs of \( \text{HwCHK}() \).\(^4\) Also note that sensor nodes are increasingly equipped with hardware acceleration for cryptography, such as many ZigBee-compatible wireless controllers, while secure memory in face of physical attacks is a real cost factor.

Moreover, more efficient solutions that a full Diffie-Hellman key exchange are possible. For example, the verifier \( V \) can generate an ephemeral RSA key pair with short public key exponent. The exponent and RSA modulus are transmitted to the prover \( P \) encrypted under \( r_N \). The prover can then recover the public key and modulus to send an RSA-encrypted, randomly chosen session key \( K \) to \( V \). This scheme provides forward secrecy since the adversary needs the RSA secret key to recover \( K \). This approach is similar to Encrypted Key Exchange \[50\], where a Diffie-Hellman public key is authenticated by

\(^4\) In contrast, the main attestation protocol requires the prover \( P \) to send only the final checksum response \( r_N \) and the main cost for \( P \) is likely checksum computation itself, which by design must be computationally expensive.
CHAPTER 7. PUF-BASED REMOTE ATTESTATION

encrypting it under a short user password\(^5\).

7.4.3 Verifier Authorization

Another important aspect is the problem of authorizing the verifier. Since remote attestation can consume significant resources and disclose potentially sensitive information about the system state, it is important that only particular authorized parties can request such an attestation.

In PUF-based attestation, this can be achieved with a simple challenge-response protocol based on the secret CRPs that are already shared between prover and verifier. For instance, let the verifier \(V\) contact the prover \(P\) for attestation. Instead of immediately launching the attestation, \(P\) may first issue an authorization challenge \(ac\) to \(V\), and \(V\) can then select an arbitrary known CRP \((x, y)\) from its database to determine the authorization response \(ar = MAC(y, ac)\). \(V\) can then authorize the actual attestation challenge \(c\) to \(P\) by including the token \(ar || x\). This will enable \(P\) to obtain \(y \leftarrow PUF(x)\) and validate that \(ar = MAC(y, ac)\) before launching the actual attestation, ensuring that the attestation challenge originates from a device that knows at least one CRP of \(P\).

Note that although the adversary considered for our attestation protocol may know a number of valid CRPs, these become known only by compromising \(P\) and directly querying the PUF. However, in that case the verifier authorization is rendered moot as the adversary can perform any kind of DoS or battery-starving attacks directly on \(P\).

7.4.4 Cooperative Attestation

As a last modification, we consider the problem of performing precise time measurements in the presence of network delays. In wireless sensor networks, hop-to-hop communication combined with energy restrictions at the sensor nodes used for routing can impose high transmission delays and more importantly jitter. Previous attempts to solve this issue are using trusted co-processors \([186, 268]\). However, we think a more practical and interesting solution is to leverage node collaboration as suggested in \([335]\).

In particular, our protocol allows delegating the time measurement to the direct neighbor of the prover without modification. The measuring neighbor only needs to record the response time and forwards it to the verifier in an authenticated fashion, together with the original response of the prover. Multiple neighbor nodes in the same broadcast zone can collaborate in this action and inform the base station about their measured time, resulting in a threshold-secure attestation scheme.

To attest the whole sensor network, this process starts at the verifier’s direct neighbors and iteratively covers all nodes in the network. We must emphasize that the security of this approach also depends on the integrity of the neighbors of \(P\), i.e., the time it takes the adversary to compromise a given set of nodes after attestation. However, if the iterated attestation is executed as an attestation payload, the attack surface remains minimal.

\(^5\)An information leakage attack is known against the encrypted Diffie-Hellman key in the original scheme \([50]\). This partition attack can be mitigated by proper encoding and choice of the key RSA-key pair. Additionally, such an attack would be less effective in our case, where the encryption key \(r_N\) is a strong secret used in only one session.
7.5 Security Considerations

In the following we argue that our presented solution achieves the security goals of software attestation described in Section 7.2. For this purpose, we assume that secure software attestation schemes and PUFs exist (see Section 7.1).

7.5.1 Correctness and Unforgeability

Existing software attestation schemes consist of two main procedures, Gen() and CHK(), which are iteratively executed to compute the checksum $r_N$. Our proposed modifications to this algorithm are limited to the Gen() procedure, where we add the intermediate PUF responses $y_{i-1}$ as additional randomizing input to the memory address computation. Hence, to guarantee that our changes do not affect correctness and unforgeability (see Section 7.2), we have to ensure that the output distribution of both the original and our modified Gen() procedure are computationally indistinguishable.

In practice, Gen() is typically implemented by a simple PRNG, which maintains an internal state that is (partly) used to generate pseudo-random outputs. The PUF response $y_{i-1}$ must be incorporated into the PRNG such that all its subsequent outputs depend on $y_{i-1}$ but without destroying their pseudo-randomness.

In case of the Gen() function of the SCUBA protocol (Algorithm 2), the PRNG is implemented as a T-function that, starting from a random seed, generates a pseudo-random sequence of 16 bit values. To incorporate the PUF response $y_{i-1}$ into the PRNG, we can simply add $y_{i-1}$ to the state of the T-function. Note that due to the unpredictability of the PUF (Section 7.1), $y_{i-1}$ is a pseudo-random value. Hence, adding $y_{i-1}$ to the state of the T-function is equivalent to reseeding the PRNG. However, if the PRNG is secure, insertion of additional entropy will not modify the statistical distribution of its output.

7.5.2 Prover Authentication

The main security goal of our design is to link the checksum to the hardware of a prover $P$. Our solutions achieves this by authenticating $P$ based on the outputs of its hardware checksum $HwCHK()$. For this purpose, we must simply ensure that a sufficient amount of authenticating information is generated by the hardware checksum $HwCHK()$ and incorporated into the attestation checksum $r_N$ to prevent simple guessing attacks, i.e., $N \cdot l_{y_{i-1}} \geq 80$ bits, where $N$ is the total number of $HwCHK()$ iterations and $l_{y_{i-1}}$ is the bit length of $y_{i-1}$.

7.5.3 Prover Isolation

The most challenging problem is to prevent the prover $P$ from delegating (part of) the software checksum computation to other systems. In our design, we run the soft- and hardware checksum in parallel to create a large algorithmic interdependence between them. In the following, we derive constrains for the minimum amount of data that must be exchanged between hard- and software checksum to assure significant slowdowns in case of a delegation attack.
To detach the computation of the software checksum from the hardware of the prover, the adversary must simulate the hardware checksum output $y_i$ in each iteration of the attestation algorithm to generate the correct input to the $\text{Gen}()$ procedure, and the intermediate checksum values $r_i$ used as input to the hardware checksum. There are two major obstacles for the adversary to do this within the time frame allowed for successful attestation: (1) The adversary must involve the original hardware checksum procedure on the prover’s hardware since this cannot be simulated or copied to another device due to the unpredictability and tamper-evidence (i.e., the unclonability) of the PUF. (2) The access speed of the hardware checksum is limited by its physical design and cannot be increased due to the tamper-evidence of the PUF.

Hence, an adversary that outsources the checksum computation to an external device is always limited by the maximum throughput of the prover’s external communication interfaces when transferring the inputs and outputs $x_i, y_i$ of the hardware checksum, which in turn are required in each iteration of the software checksum computation.

We assume a strong adversary that can calculate the simulated software checksum instantly. Such an adversary is only limited by the time $t_{x_i}$ and $t_{y_i}$ it takes to transmit $x_i$ and $y_i$, respectively, plus the access time $t_{\text{HwCHK}}$ of the hardware checksum: $t_A = t_{x_i} + t_{y_i} + t_{\text{HwCHK}}$. In contrast, the time for an unmodified checksum computation is limited by the computational speed of the legitimate hardware and software checksums, which are executed concurrently and thus have similar access time\(^6\): $t_P = t_{\text{CHK}} = t_{\text{HwCHK}}$, where $t_{\text{CHK}}$ is the access times of the software checksum. Hence, the security of our scheme depends on the amount of information that is transferred between hard- and software checksum during the computation of the checksum $r_N$, and the time it takes to transfer this data over the accumulated transfer capacity of all external communication interfaces (ExtBps) of the prover:

$$t_A - t_P = t_{y_i} + t_{x_i} = \frac{N \cdot (l_{x_i} + l_{y_i})}{\text{ExtBps}}$$

In Section 7.3.2 we propose optimizations that work with reduced bit lengths for $y_i$ and reduced entropy of $x_i$. The impact of reduced bit length of $y_i$ is straightforward to assess by re-evaluation of the above security margins. However, the reduction of the challenge space changes may allow the adversary to predict the responses of the PUF in a similar way as the procedure used for on-demand generation of CRPs: After reception of the offset $q$, the compromised prover $P$ is asked to generate the complete CRP subspace for the current attestation and to return only the responses, ordered by the corresponding challenges. In this case, the time $t_A$ that the adversary takes to calculate the final software checksum value $r_N$ is limited only by the time required to generate and transmit all hardware checksum outputs $y_i$, i.e., $t_A = 2^{l_{x_i}} \cdot (t_{\text{HwCHK}} + t_{y_i})$. The resulting timely advantage for the adversary can be derived as

$$t_A - t_P = (2^{l_{x_i}} - k) \cdot t_{\text{HwCHK}} + 2^{l_{x_i}} \cdot t_{\text{CHK}}.$$

As we can see, the adversary’s timely advantage critically depends on the length of $x_i$. Since we aim to keep $t_{y_i} = l_{y_i} / \text{ExtBps}$ small to speed up on-demand generation and transmission of new CRP subsets and since $t_A$ grows exponentially with increasing $l_{x_i}$, we

\(^6\)See Section 7.4.1 on how to synchronize the performance of hard- and software checksum.
can set \( l_y = 1 \), and use \( l_x \) as adjustable security parameter.

For instance, the Telos Mote used for the prototype implementation of SCUBA [272] supports a wireless interface with at most 250 kbps and a wired UART over USB interface with (typically) 115.2 kbps. Considering our example instantiation based on SCUBA for \( l_x = 20 \) bits, the adversary would need to transmit \( l_x + l_y = 160 + 20 \) bit for \( r_i \) and \( y_i \), respectively, in each iteration. The prototype implementation of [272] computes 40,000 checksum iterations in less than 2.87 seconds. This requires the adversary to transmit \((160 + 20) \cdot 40,000 = 7,032\) kBit per attestation, incurring a significant overhead of 7,032 kBit/(250 kbps + 115.2 kbps) = 19.26 seconds.

Hence, if the implementation uses CRP sub-spaces of size \( l_{f(r_i)} = 20 \) bits and a \texttt{HwCHK()} that operates at 14 kHz, an adversary, who aims to query the complete CRP sub-space in a batch instead of executing software and hardware checksum in parallel, takes 74.85 seconds, which is a massive overhead compared to 2.85 seconds of the legitimate computation time.

### 7.6 Related Work

Purely software-based attestation was presented in [275] as response to a previous idea on using hardware side-effects in a timed checksum computation [171]. Since then, several works investigated the extension and optimization of software-based attestation for different platforms [273, 113, 272, 192, 121, 277].

Some approaches propose to delegate the time measurement to previously attested neighbors of the device to be attested [275, 335]. However, these concepts exclude hardware manipulation and physical replacement of prover devices by assumption, and thus solve the problem we consider here only in a very limited sense.

Multiple works also consider the extension of software attestation with hardware trust anchors like TPMs and SIM-cards [268, 160, 185], so that the critical timing measurement and prover authentication can be performed directly at the prover by signing the attestation challenge and response once they are available. However, as pointed out earlier the integration of complex secure co-processors seems to contradict the idea and purpose of software attestation. Moreover, it is unclear why an integrated TPM or trusted SIM card cannot be used directly as a trust anchor in TCG-style attestation. In contrast, our proposed integration of a PUF as low-cost trusted PRF results in a much more lightweight construction and may also provide a basic tamper-evidence that is often desired in low-cost embedded systems.

A recent proposal for lightweight attestation employs a modified memory access control and a measurement routine in ROM [88]. The design was further analyzed and improved in [109, 110]. Chapter 9 picks up and generalizes these concepts to provide generic isolation of software modules.

### 7.7 Summary

We presented a novel approach to attest both the software and the hardware configuration of a remote, low-cost embedded device that does not possess trusted hardware compo-
ments. Our solution combines existing software attestation with cost-efficient physical security primitives, i.e., Physically Unclonable Functions (PUFs), and resolves some of the fundamental limitations of purely software-based attestation. In contrast to existing software attestation protocols, our scheme does not assume an authenticated channel between the prover and the verifier and reliably prevents remote provers from colluding with other devices to forge the software checksum.

A major limitation of PUF-based attestation remains the overall complexity and unusual assumptions of software-based attestation protocols (see Chapter 6). In particular, the concept of temporarily using the complete computational resources of the prover is problematic for resource-constrained embedded systems where performance and power-efficiency are critical. Another limitation is that PUF-based attestation is inherently symmetric, i.e., based on shared secrets which limit the scalability and preclude features such as non-reputability. We will present an alternative architecture in Chapter 9 which supports greater flexibility, including public key cryptography.
Chapter 8

Transaction Security with SoftCards

In recent years, consumer hardware has been extended with the ability to execute programs independently from previously loaded software, including the main operating system [154, 24, 35]. These Trusted Execution Environments (TEEs) allow the secure re-initialization of the system state at runtime, and to launch a new OS or simply to execute a small security service without the main operating system an applications being able to interfere. Sophisticated systems have been proposed to use these capabilities for securing passwords and online transactions. They move authentication and authorization procedures of applications into the trusted environments and use the complete hardware control to assure that any entered user passwords originate from a physically present user and cannot be intercepted by malware [108, 210, 64]. However, all these approaches assume a substantial modification of existing authentication procedures and/or software stacks. As a result, the resulting prototypes are limited in scope and provided functionality, and have not found any significant adoption.

In this chapter, we present the design and integration of a software security token that uses trusted execution technology available in commodity PCs. Our solution achieves comparable security features to hardware tokens in face of software attacks and basic protection against common hardware attacks such as device theft and offline brute-force attacks. In contrast to previously proposed secure transaction systems using trusted computing, the solution we present in Section 8.1 aims at easy interoperability and deployment, supporting the widely accepted PKCS#11 standard. Hence, our prototype is directly usable with many existing applications, such as enterprise single sign-on solutions, authentication in VPN, e-Mail and WiFi clients and password managers. Leveraging facilities for secure user input/output, we also implement secure and convenient mechanisms for networked deployment, backup and migration of software tokens in Section 8.2. In Section 8.3 we sketch a prototype implementation based on Flicker and OpenCryptoki. The complete documentation and source code of our prototype is available online, providing a free open-source prototype that can be easily deployed and extended for further research: http://code.google.com/p/softcards/.
8.1 Softer Smartcards with Trusted Execution

Although available for over a decade, cryptographic security tokens with asymmetric multi-factor authentication are still not in common use for many daily authentication procedures. Despite their significant advantages, service providers and users still prefer more usable but weak password-based authentication protocols that are highly vulnerable to simple malware and phishing attacks.

In the following we present the security requirements and high-level architecture of a software-based smartcard. Our software smartcard (or SoftCard) is secured using trusted execution environments instead of the traditional secure co-processor module, thus enabling most functions to be flexibly defined in software. Leveraging the PKCS#11 cryptographic token interface, our solution is backwards-compatible with many existing applications and extensible to secure complete user transactions with secure user I/O.

8.1.1 Requirements and Adversary Model

We identify the following security requirements for a secure cryptographic token:

Secure Token Interface: The interface used for interaction between the (trusted) token and the (untrusted) outside world must prevent the leakage or manipulation of any secret information contained in the token, such as keys or key properties.

Secure User I/O: The user interface of the token solution must provide a secure mechanism for the user to (1) judge the current security status of the token, (2) infer the details of a pending transaction to be authorized and (3) input the authorization secret (Personal Identification Number, PIN).

Moreover we can identify the following functional requirements:

Interoperability: The token should use a standards-compliant interface to provide interoperability with existing applications, such as the PKCS#11 or Cryptographic Service Provider (CSP).

Usability: The token should be usable in the sense that its use should not impose a significant burden on the user. In particular, the user should not be required to buy additional hardware and the token should be usable in mobile scenarios.

Migration and Backup: The token should provide facilities for secure migration between different hardware platforms or implementations, and for the creation of secure data backups.

Adversary Model. We assume an adversary that can compromise the user’s operating system and applications (malware, trojan horse attacks) or lure the user into revealing authentication secrets by imitating local applications and web services (phishing, spoofing). However, the adversary has no physical control over the user’s cryptographic token and is thus restricted to the application interface of the token. The goal of the adversary
is to compromise long-term secret keys, to authorize transactions or decrypt confidential data of the user.

This adversary model captures common attacks such as malware and phishing attacks that commodity computer users are facing today. Our evaluation shows that software-based smartcards meeting the above described requirements can provide a similar security protection to traditional hardware smartcards, except for the relatively risky and less dominant physical hardware attacks.

### 8.1.2 System Architecture

We aim to protect the user’s sensitive information by implementing the functionality of a cryptographic token in a Trusted Execution Environment (TEE). The TEE manages and protects the credentials even if the underlying operating system is untrusted and potentially compromised. Hence, the trusted execution environment must also establish a trusted input/output path between user and token and must be able to authenticate itself towards the user. We combine the TEE with the data sealing mechanism of the TPM to let only the locally authenticated user access the secret token state, and to assure that only a trusted token implementation inside the TEE can access the sensitive token state.

![Figure 8.1: Legacy OS integration of an TEE-based cryptographic token.](image)

The overall architecture of our TEE-based token is depicted in Figure 8.1. It consists of a “legacy” software stack with operating system, token middleware, and the trusted execution environment on the left-hand side, which runs our Secure Execution Token Application (SETA) in parallel to the legacy operating system.

Applications must be compatible with the middleware interface to make use of cryptographic tokens. Multiple tokens can be supported by the token middleware, and in our design we add another type of token beside hardware and software tokens that calls the TEE with appropriate parameters. The call into the TEE is implemented using an TEE driver provided by the legacy OS, such as Flicker. Finally, the SETA is started as the main component of this architecture, implementing the semantics of the cryptographic token and enforcing the security-critical interfaces based on (1) isolated execution using the TEE and (2) state-specific data encryption using the TPM sealing operation (cf. Section 8.5).
The SETA component running in the TEE supports three major interfaces, as shown in Figure 8.1. The user interface (user I/O) implements interaction with the platform peripherals for user input and output using keyboard and graphics card. The TPM interface is used for basic interaction with the TPM, specifically for the sealing, unsealing and interaction with the TPM’s monotonic counters. Finally, the input and output of the SETA module, upon entering and leaving the TEE, is used to implement the cryptographic token interface. The token interface operation \( op() \) that is to be executed is provided together with the encrypted token state \( ts \). SETA must then request authorization for executing \( op() \) by requesting the PIN \( pin \) via user I/O, decrypt the token state \( ts = \text{unseal}(pin, [ts]) \) using the TPM interface and compute the result \( res = \text{execute}(op(), ts) \) to be returned as the output of SETA.

8.2 Component Interaction and Protocols

Usability and flexibility are among the main concerns when using cryptographic tokens today. In the following, we present more detailed protocol flows for the main smartcard operation and caching of the user’s PIN, and discuss the problems of deployment, migration and backup.

8.2.1 Deployment

Before using the software token for the first time, the user must initialize it in a trusted enrollment environment and choose an individual picture \( img \) and PIN number \( pin \), as illustrated in Figure 8.2(a). The picture is used later on to authenticate Secure Execution Token Application (SETA) towards the user, while the PIN is used to authenticate the user towards the token. After initialization, the token state is sealed to the expected TPM PCR values of SETA, which are also supplied as input, so that only SETA can open and modify the token.

Note that if the initialization and enrollment system is different from the user’s target platform, the TEE-based token can be created centrally and deployed to the user’s target platform using the token migration procedure described in Section 8.2.2. This is particularly interesting for enterprises, which often require central enrollment and backup of authentication and encryption secrets. Hence, the overall procedure remains identical to the deployment of regular smartcards, except that the user is not required to (but could) use a physical device to transport the token state from enrollment platform to the target platform.

8.2.2 Migration and Backup

Secure backup and migration of credentials is essential for both enterprise and personal use. Note that backup and migration are very similar, since backup can be regarded as a “migration into the future”, using either a dedicated backup platform as intermediate migration target or a direct migration to a target platform of yet unknown configuration.

For best usability in personal usage scenarios, we propose to realize secure migration simply based on trusted user I/O and a user passphrase \( K_m \) as illustrated in Figure 8.2.1:
To migrate the token, the user must first launch SETA and enter the correct PIN \textit{pin} to access the protected token state \textit{ts}. Upon issuing the \texttt{migrate()} command, SETA encrypts \textit{ts} using a symmetric randomly chosen passphrase \textit{K}_m and returns the encrypted token state \([ts]_{K_m}\) to the untrusted environment of the source platform. The passphrase \textit{K}_m is disclosed to the user via trusted I/O, enabling the owner of the token (authorized by the entered PIN) to import the encrypted token state \([ts]_{K_m}\) at the target platform.

Additionally, the migration procedure can be wrapped in a trusted channel if the target platform is known in advance and is equipped with a TPM: As proposed in [42, 7], the encrypted token state \([ts]_{K_m}\) can be bound to a specific target platform’s TPM and platform state before disclosing it to the source platform’s untrusted environment. As a result, the protected token state can only be imported by the designated target platform, and only from within the trusted SETA environment. Token migration using trusted channels can thus effectively mitigate brute-force attacks on \textit{K}_m and, based on the authentication image \textit{img} sealed to SETA, also prevent impersonation of the importing application at the target platform. Note that if the backup system is equipped with a TPM, this extended migration protocol can also be used for secure backups.

### 8.2.3 Token Operation

The protocol for the regular operation of our TEE-based token is illustrated in Figure 8.3. As outlined in previous Section 8.1.2, the token middleware executes security-sensitive operations \texttt{op()} on behalf of the applications, which in turn delegates execution of the required algorithms to SETA and returns the result \textit{res} at the very end of the protocol flow. For this purpose, the middleware keeps track of one or more token states \([ts]\) and their supported types of operations \texttt{op()}. Specifically, \([ts]\) consists of encrypted token state \textit{cstate}_{tk}, \textit{cstate}_{pal} and the corresponding encryption keys \([K_{tk}], [K_{pal}]\) sealed to the SETA environment. To execute a specific operation \texttt{op()}, the middleware determines the required cryptographic token algorithm \textit{algo} to be executed and then asks the operating system’s TEE driver to launch an instance of the SETA module, supplying the algorithm identifier \textit{algo}, the token identifier \textit{id} and the respective protected token state \([ts]\) as
CHAPTER 8. TRANSACTION SECURITY WITH SOFTCARDS

TPM

untrusted CPU mode

secure CPU mode

User

1. op()

2. \([K_{pa}], [K_{pal}], dist, cstate_{pal}, cstate_{pal}, algo, id\)

3. \(K_{pal} \leftarrow \text{unseal}([K_{pal}], \ldots)\)

4. verify\(\text{ctr}\)

5. \(\text{pin} \leftarrow \text{userAuth}(\text{img})\)

6. \(K_{pal} \leftarrow \text{PBKDF}(\text{pin}, \text{id})\)

7. \((\text{cstate}_{tk}, \text{ctr}_{tk}) \leftarrow \text{exec}(\text{algo}, \text{cstate}_{tk})\)

8. \(\langle \text{res}, \text{cstate}_{tk} \rangle \leftarrow \text{exec}(\text{algo}, \text{cstate}_{tk})\)

9. res

Figure 8.3: System component interaction during normal operation.

shown in step 2.

Once launched, SETA unseals \(K_{pal}\) in step 3 to decrypt the PAL meta-data state \(cstate_{pal}\) to retrieve the master key \(K_{ctr}\) of the counter list \(dist\) and the secret authentication picture \(img\). \(K_{ctr}\) is used to decrypt \(ctrs\), a list of counters that is shared by all trusted applications on the platform. In correspondence with previous work on secure virtual counters [266], the sum of the elements in \(ctrs\) is compared with a predefined secure monotonic counter in the TPM in step 4. If the sum matches, \(ctrs\) and thus also \(cstate_{pal}\) and the individual token’s counter \(ctrs[\text{id}]\) and \(img\) are fresh. In step 5, the picture \(img\) is used to authenticate SETA to the user and retrieve the authentication PIN \(\text{pin}\). The PIN and token identifier \(\text{id}\) are fed into the password-based key derivation function (PBKDF2 [166]) to generate the user authentication key \(K_{pin}\). This secret is in turn used in step 6 to unseal \(K_{tk}\), so that \(cstate_{tk}\) can be decrypted to retrieve the secret token state \(cstate_{tk}\) and the verification counter \(cntr_{tk}\). To assure the freshness of \(cstate_{tk}\), SETA checks if known fresh \(cntr\text{id}\) is equal to \(cntr_{tk}\). If successful, the actual token algorithm \(\text{algo}\) can finally be executed in the context of \(cstate_{tk}\), yielding the return value \(\text{res}\). If the state \(cstate_{tk}\) was updated during the execution of \(\text{algo}\), we must increment the freshness counters \(cntr_{tk}\) and \(ctrs[\text{id}]\) (step 7), update the TPM hardware counter accordingly and then re-encrypt the token state \(cstate_{tk}, cstate_{pal}\) (dashed box). If updated, the new states \(cstate_{tk}, cstate_{pal}\) are returned together with the result \(\text{res}\) in step 8. Finally, the result of the operation \(\text{op}()\) can be returned to the application in step 9.

Note that even if verification of the virtual counter vector \(ctrs\), which is shared together with \(K_{ctr}\) among all trusted applications that require secure TPM-bound counters, is unsuccessful, the application can still recover the desired secret states and also determine the number of version rollbacks that have occurred as \(\text{num} = cntrID - cntr_{tk}\). Hence,
in case of system crashes or misbehavior of other software components SETA can inform the user and offer recovery. However, in this case the user must assure that no malicious version rollback of the sealed token state \([ts]\) took place.

While the TPM specification imposes a frequency limit on the use of the TPM’s secure monotonic counters, it is unlikely that the use of SETA is affected by this limit: Most common operations carried out with the token, such as signature creation, do not modify the token state and thus do not require an update of the TPM secure counters. Moreover, common operations such as enquiring the algorithms supported by the token are not actually security sensitive and can be implemented within the token middleware’s SETA adapter.

### 8.2.4 PIN Caching

It is often useful to cache a given user authorization for a certain time or certain number of uses. For example, in the current adoption of security tokens in health care systems it is often simply too time consuming to authorize prescriptions and other documents of minor sensitivity individually. Hence, so-called “batch signatures” were introduced that sign multiple documents at once [106]. In the following, we present an optional PIN caching mechanism for our TEE-based token that allows the authorization of multiple token operations.

Instead of requiring the PIN for each transaction, our system is able to securely cache the PIN for multiple uses. For this purpose, we seal the cached authorization secret \(K_{pin}\) to the trusted system state of SETA and add a usage counter \(uses\) to be maintained by the (untrusted) token middleware. We verify the value of \(uses\) based on the non-invertible status of a PCR register \(p\), so that the usage count can be tracked independently from the token state \(cstate_{tk}, cstate_{pal}\). Another advantage of this construction is that an unexpected reset of the platform or update of the PCR \(p\) does not invalidate the token state but only the cached PIN.

Figure 8.4 shows a modified version of the main protocol in Figure 8.3 to support PIN caching. When the PIN is provided for the first time and should be cached, the maximum desired number of PIN uses \(uses_{max}\), the user authorization secret \(K_{pin} \leftarrow PBKDF(pin, id)\) and a randomly chosen secret \(s\) are added to the token state \(cstate_{pal}\). For subsequent SETA executions with cached \(K_{pin}\), the respective values are recovered from \(cstate_{pal}\) as shown in Figure 8.4 after step 2.

In step 4, the current value \(reg'\) of PCR \(p\) is read in addition to the verification of \(ctrs\). Due to the non-invertibility of the PCR states, this allows verification of the value of \(uses\) based on the purported PCR pre-image \(reg\) and the secret \(s\). If this verification succeeds and \(uses < uses_{max}\), the cached PIN can be used and the PCR \(p\) is updated for the incremented \(uses\) counter in step 5a. Otherwise, the user is asked for authorization in step 5b. After successful execution of the following steps 6 and 7, the result \(res\) is returned together with the current usage counter \(uses\) and possibly updated state \([ts]\) in step 8 and 9.

If step 6 executes successfully following step 5b, the caching state can be reset as \(reg = reg', uses = 0\). Otherwise, if the PIN was (repeatedly) entered incorrectly, \(K_{pin}\) should be purged from \(cstate_{pal}\). As a result of this construction, the PIN caching works
mostly independent from the more expensive token state updates: The values of $uses_{max}$ and $s$ can be considered relatively static and remain in $cstate_{pal}$ even if PIN caching is not used. Moreover, an unexpected modification of PCR $p$, including platform reset, only invalidates the PIN caching status, requiring only a regular user authentication to continue operation.

8.3 Implementation and Results

We implemented a proof of concept prototype of our TEE-based token based on the software token of the OpenCryptoki middleware [44]. OpenCryptoki implements the PKCS#11 interface [253], a well-known standard supported by many applications, including major web browsers, eMail clients, several VPN clients and other authentication solutions.

To build SETA, we separated the security-sensitive functions implemented by the OpenCryptoki software token into a separate software module that is then executed using the Flicker TEE driver. Additionally, we implemented basic drivers for accessing keyboard and graphics hardware from within SETA, to provide a secure user input/output interface while the untrusted OS and applications are suspended by the TEE. In this respect, we extend the work of [108] that could only provide a basic text output from within the TEE.

8.3.1 Performance Evaluation

While the processors in many hardware tokens are optimized for low cost and resistance against physical attacks, our solution benefits from the high performance of today’s CPUs.
The main delay for executing operations with SETA is caused by the switch into the TEE itself using Flicker, and the interaction with the (often rather slow) TPM for unsealing and counter update.

<table>
<thead>
<tr>
<th>Existing Tokens</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrypToken MX2048</td>
<td>2.9s</td>
</tr>
<tr>
<td>eToken Pro 32k</td>
<td>4.3s</td>
</tr>
<tr>
<td>opencryptoKi SW-Token</td>
<td>0.05s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Our SETA Solution</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch to TEE</td>
<td>+1.20s</td>
</tr>
<tr>
<td>TPM interaction</td>
<td>+1.50s</td>
</tr>
<tr>
<td>Signing in TEE</td>
<td>+0.05s</td>
</tr>
<tr>
<td>Overall Signing in SETA</td>
<td>2.75s</td>
</tr>
</tbody>
</table>

Table 8.1: Speed comparison for a PKCS#11-compliant 1024 bit RSA signature.

We compared the time required for a PKCS#11 signing operation using hardware and software tokens versus using our SETA. The signature operation is perhaps the most common operation for security tokens, as it is used for authentication as well as document signing. The specific document or data length is insignificant in this case, as most applications simply hash the document themselves and only sign the hash digest, to reduce the amount of data that would otherwise have to be transferred and handled by the token.

Specifically, we compared an older Aladdin eToken Pro 32k and a newer MARX CrypToken MX2048 as hardware tokens against the opencryptoKi software token and our TEE-based solution on a standard Dell Optiplex 980 PC with an Intel 3.2 GHz Core i5 CPU. For the signature operation we use the PKCS#11 C_Sign command using RSA-1024 as the signature mechanism. As can be seen in Table 8.1, SETA is almost twice as fast as the older eToken Pro and still faster than the modern MX2048.

Unfortunately, since the legacy OS is suspended while SETA execution, its overall execution time still incurs a significant disruption for the user and other software. However, observe that in the breakdown of the SETA timings in Table 8.1, the switching into TEE and TPM interaction incur by far the most delay (98%). Hence, other TEE implementations such as ARM Trustzone may still enable a faster switching into the TEE context and also provide the option to maintain a transient state between calls into the TEE, such that these delays can be reduced significantly.

8.4 Security Considerations

The security of our overall scheme depends on the enforcement of information flow control to maintain the confidentiality and integrity of the internal token state. Specifically, our token must meet the two security requirements formulated in Section 8.1, (1) preventing unauthorized leakage or manipulation of the token state and (2) providing a user interface that is secure against spoofing or eavesdropping attacks by a compromised legacy operating system and that detects attempts to tamper with the SETA token implementation.

Finally, we discuss the resilience of software tokens compared to traditional cryptographic hardware tokens.
Secure Token Interface. Requirement (1) holds based on the assumption that the user and TPM are trusted and the PKCS#11 token interface is secure and securely implemented. The first two assumptions are standard assumptions and differ from the security of regular hardware-based cryptographic tokens mainly in that the TPM is not designed to be secure against hardware attacks.

Considering the limited security of hardware tokens against hardware attacks [32, 294, 224] and the prevalence of remote software attacks it is reasonable that we exclude hardware attacks in our adversary model. While some attacks on PKCS#11 implementations have been shown [57], the specification itself is considered highly mature and implementation flaws are independent from the token type.

Secure User I/O. Requirement (2) is met by our combination of TEE and TPM, which results in isolated execution of trusted code with full control over the local platform. The TEE suspends the legacy OS, preventing any potentially loaded malware from manipulating the execution of SETA payload and giving it full hardware access. By implementing appropriate keyboard and graphics drivers in SETA we can thus provide secure I/O on standard computing platforms. Additionally, to prevent the replacement of SETA by malicious applications, we use the TCG TPM’s sealing capability to bind data to designated system states, such that a malicious SETA’ ≠ SETA is unable to access the protected token state \( ts \) and user authentication image \( img \). Specifically, since only the pristine SETA program can access and display the secret authentication image \( img \), the user can always recognize if the currently running application is the untampered SETA. A well-known open problem, in this context is that users often disregard such security indicators, allowing an adversary to spoof the user interface and intercept the secret PIN [267]. However, in our solution, an attacker that has gained knowledge of the user’s PIN also requires the untampered SETA program to which the user’s sealing key is bound and to which he thus has to enter the PIN (physical presence). Hence, although our solution cannot fully prevent interface spoofing against attacks against unmindful users, the attack surface is notably reduced by restricting the TPM unseal operation to pre-defined physical platforms and requiring physical presence. We suggest that enterprises monitor the migration and authorization of SETA modules for such events.

Comparison with Hardware Tokens. The security of our design is based on the security of TCG TPM operations, the TEE and the PKCS#11 interface. The TCG TPM provides protection against PIN brute-force attacks, while the TEE prevents interference of malware, protecting the user input and output against eavesdropping and manipulation. However, in contrast to cryptographic hardware tokens, the TCG TPM is not designed for security against hardware attacks. In fact, current implementations were demonstrated to be vulnerable to simple hardware attacks [283]. Our Trusted Execution-based solution is thus generally less secure against hardware attacks than cryptographic hardware tokens. However, hardware attacks are also considered the strongest possible attack on security systems and even tamper resilient hardware is not completely secure against them [32]. Common hardware attacks can be mitigated when the TPM becomes integrated with the CPU or mainboard chipset and by implementing basic algorithmic countermeasures against side-channel analysis.
When considering software attacks, the PKCS#11 interface may leak information, e.g., by allowing the export of private keys or the time taken for a specific transaction. Using a model checker, several such implementation flaws were recently identified in cryptographic hardware tokens and also in the OpenCryptoki software [57]. Note that while such vulnerabilities affect both, hardware and TEE-based tokens, they are easier to patch in SETA, and the TPM sealing and version rollback functionality prevent the use of older software versions and/or previous encrypted state files.

Some recent works also manage to break the rather novel TEE implementations through security bugs in BIOS and PC firmware [328, 329]. The works show that PC-based TEEs currently still require secure BIOS implementations, which can be verified using the respective TPM PCRs. Again, these vulnerabilities in the execution environment are not specific to our solution, as illustrated by recent attacks on dedicated smartcard readers \(^1\). However, similar to bugs in the PKCS#11 implementations such vulnerabilities are rare and usually very hard to exploit in comparison with common trojan horse or phishing attacks.

### 8.5 Related Work

Several previous proposals aim to protect user credentials, typically using either a persistent security kernel that protects the credentials [118, 156, 45] or relying on external trusted hardware [161]. We know of only one work that uses current TEEs to provide secure transactions using secure user I/O [108]. They use Flicker to take control of devices and then ask the user to confirm a specific transaction. Using remote attestation, a remote system can then verify that the transaction was securely confirmed by a human user. All of these approaches require substantial modification of individual applications or even the addition of a hypervisor. In contrast, our solution uses the widely established PKCS#11 interface [253] and works seamlessly with many existing applications and operating systems that use this interface.

The On-board Credentials (ObC) framework was introduced for mobile embedded devices [184]. ObC uses device-specific secrets of M-Shield to provide an open provisioning protocol for credentials (code and data), that are securely executed/used inside the TEE. TruWalletM [65] implements a secure user authentication for web-services based on ObC. ObC focuses on credential management and is thus complementary to our solution, which focuses on making the actual online transaction secure for many existing applications.

### 8.6 Summary

We introduced an TEE-based PKCS#11 token that combines the flexibility of software credentials with the security of modern trusted computing technology. While our solution does not achieve the same resilience against hardware attacks as cryptographic hardware tokens, it presents a significant improvement over software-based solutions or

\(^1\)E.g., a smartcard reader by Kobil allowed unauthorized firmware manipulation: http://h-online.com/-1014651
cryptographic tokens used without dedicated keypad and display. Using secure user I/O and the PKCS#11 cryptographic token interface standard, we provide a secure plug-in solution that is especially attractive in today’s mobile computing environments and fast evolving software and security solutions.

One of the main conclusions to draw from our proof-of-concept demonstrator is that the delay and impact of launching an TEE on current platforms is still too high. In particular, the currently employed hardware solution imposes a significant interruption of the main OS execution, including sensitive hardware drivers such as WiFi connection. Ideally, TEEs should run in parallel to the existing software stack, isolating access to hardware resources on demand.
Chapter 9

The TrustLite Security Architecture

9.1 Introduction

In the emerging “Internet of Things” with its increasingly complex embedded systems, low-cost resource-constrained devices play a critical role for personal, economical and national security. As these systems fundamentally rely on the data they collect and produce, the assurance of end-to-end protection of data and software integrity is critical for security and scalability. Unfortunately, while the design of cryptographic accelerators is well established, no solutions are available to trusted, isolated execution environments for fundamental services like attestation and sealing, or even third party applications to be deployed and updated in the field. Hence, security compromises remain hard to detect and recovery, if possible, often requires vendor intervention.

Some proposals have been made in the literature to realize a limited version of trusted execution and software isolation. As discussed in Chapter 6 and Chapter 7, software attestation can provide a low-cost, software-only solution for attestation and, in fact, establishes a dynamic root of trust. However, expending all computational resources for a few seconds whenever an attestation or trusted execution should be performed is impractical for many scenarios with stronger performance, latency and/or power consumption constraints. Instead, it is desirable that trusted execution environments can co-exist with regular tasks in a multitasking environment, enabling low-latency context switches and secure user interaction (cf. Chapter 8).

In this chapter we present TrustLite, a security architecture for low-cost embedded systems that allows OS-independent isolation of specific software modules with different trade-offs for assurance vs. flexibility. This allows manufacturers and security providers for the first time to implement custom security services such as remote management, secure updates, remote attestation and authentication services for a broad range of devices and independently of the respective deployed OS and application software. This is particularly interesting for low-cost embedded devices, where the hardware and software environment is highly dynamic, the cost of secure operating systems or hardware protection is a significant factor and a case-by-case evaluation of the individual target platforms is prohibitively expensive.

We introduce an Execution-Aware Memory Protection Unit (EA-MPU) as a generalization of recent memory protection schemes for low-cost devices. Programmed in soft-
ware, our EA-MPU allows a flexible allocation and combination of memory and peripherals I/O regions without burdening the CPU. We solve the problem of information leakage on platform reset by introducing a simple Secure Loader sequence, which can optionally also provide a root of trust for attestation and trusted execution. We also propose a modified CPU exception engine, enabling the preemptive scheduling of trusted tasks by an untrusted OS. Our architecture supports update of software and security policy in the field and works independently of the CPU instruction set, facilitating code re-use and fast deployment. A major strength of TrustLite is that it can be instantiated in several ways, providing different security features at different cost points. In the simplest setup, TrustLite may only provide a single isolated firmware security service with hard-wired memory access control, while a full instantiation can provide remote attestation and trusted execution of multiple hardware-protected userspace applications running in a preemptive multitasking environment.

9.2 Problem and Assumptions

We consider a low-cost System on Chip (SoC) in the range of 100,000 gate equivalents, including on-chip memory and basic peripherals such as timers, interfaces for external communication peripherals and possibly cryptographic accelerators. This design point matches many of the currently available and possible future low-cost embedded computing platforms such as the popular Texas Instruments MSP430 [111]. As is the case with most embedded devices, we assume that the CPU boots from a hardwired, well-known location in non-volatile memory, such as Programmable ROM (PROM), and that any peripheral access is implemented with Memory-Mapped I/O (MMIO).

Observe that, in contrast to more costly and relatively power-hungry mobile or desktop systems, the considered target platform has no hardware support for virtualization, a secure firmware runtime or enhanced security features such as trusted execution. In particular, we do not assume a secure co-processor or CPU security extensions that provide additional privilege levels and execution modes (e.g., [33, 235, 35, 134]). Instead, the class of devices considered in this work may potentially be used as part of a programmable secure co-processor or smart card.

Despite strong economic incentives to minimize development and production costs, it is desirable that such platforms offer certain security services such as remote reporting of the software, secure software updates and possibly the ability to deploy and update security-sensitive services such as authentication or e-payment systems. Current platforms do not offer this capability, and a trusted OS approach incurs high costs for security evaluation of new drivers or protocols.

9.2.1 Terminology

We denote static (machine) code and associated data and meta-data as a program. Programs are typically designed for a particular functionality of some overall application scenario. In comparison, a task describes the runtime state of a program, software or firmware, including its CPU state, call stack and other volatile data. The Trusted Com-
Putting Base (TCB) of a task is the set of components (hardware and software) that must be secure to assure the unmodified execution of that task. Tasks which are designed and believed to implement a particular security mechanism are trusted tasks and we refer to them as trustlets.

The owner of a platform is authorized to install and modify the TCB, tasks and trustlets of that platform as desired, but may not (always) be the entity that is in physical control of the platform.

### 9.2.2 Adversary Model

The adversary’s goal is to compromise trustlets on the platform which are not owned by the adversary and do not have any trustlets issued by the adversary in their TCB.

For this purpose, we assume that the adversary has full control over the untrusted OS and tasks running on the platform. Furthermore, the adversary can convince the platform owner to deploy arbitrary code in form of additional trustlets. However, trustlets and bootstrapping routine are assumed to operate correctly, i.e., their API contains no exploitable software bugs. The adversary also controls all communication with the platform and can eavesdrop, manipulate and intercept any communication messages. However, it is assumed that any deployed cryptographic mechanisms are secure.

We assume that the high levels of integration achievable with modern IC fabrication processes render chip-level invasive attacks such as tampering, on-chip bus probing, extracting keys from on-chip memory or fault injection out of scope for economically motivated attackers and that mitigations are in place against side-channel leakage through power, electromagnetic emissions or timing behavior [182, 120, 181].

### 9.2.3 System Requirements

Low-cost embedded devices are used in many different scenarios. Typical requirements are fast cold start, low power consumption and real-time processing constraints. In many cases it is desirable to update the software in the field, and the enabling of third party secure applications such as authentication and payment plug-ins is increasingly important.

For security, the platform should prevent trustlets from tampering with each other and enable software environments with a minimal common TCB. Based on our experience with practical usages of Trusted Execution Environments (TEEs) in Chapter 8 we pose the following security requirements:

**Data Isolation:** Trustlets are isolated in the sense that no other software on the platform can modify their code. Trustlet data can be read or modified by other trustlets according to the system policy.

**Attestation:** Trustlets can inspect and validate the local platform state without other software being able to manipulate the procedure.

**Trusted IPC:** Trustlets can inspect and validate other trustlets on the platform and establish a mutually authenticated and confidential communication channel.
Figure 9.1: High-level architecture of a TrustLite platform.

Secure Peripherals: Trustlets can be provided with exclusive access to platform peripherals, such that other software cannot interfere in that interaction.

In particular, secure peripherals were shown to be essential for many applications scenarios, such as providing secure user input/output for explicit confirmation of online transactions [5, 108, 342]. Additionally, based on private communication with various potential consumers of low-cost embedded security products, the following functional requirements were found to be essential for providing a practical and versatile solution:

Fast Startup: The security extensions should not significantly impact the bootstrapping delay, e.g., by having to measure large amounts of code or perform cryptographic operations at platform or trustlet initialization.

Protected State: To support fast invocation and switching between trustlets, the platform should allow trustlets to maintain a protected software state during inactivity.

Field Updates: The solution should allow updates to code, data and security policy after deployment.

Fault Tolerance: It should be possible to interrupt trustlets on unexpected errors or timeout.

9.3 The TrustLite Security Architecture

In the following we present the base components and procedures of the TrustLite security architecture and explain how they contribute to the memory protection and trusted execution of isolated software tasks.

9.3.1 Architecture Overview

The high-level hardware and software architecture of TrustLite is illustrated in Figure 9.1. The platform consists of a System on Chip (SoC) with (possibly insecure) external peripherals. Inside the SoC boundary, a CPU core is integrated with at least one PROM, alarm
timer, and some limited amount of RAM. A Memory Protection Unit (MPU) enforces access control on all memory accesses, including regular memory as well as memory-mapped device I/O (MMIO). Depending on the application, a number of additional components may be included in the SoC, including cryptographic accelerators, display adapters and communication interfaces.

On top of the trusted SoC, a Secure Loader is responsible for loading all desired trustlets and their critical data regions into on-chip memory. Additionally, it programs the MPU to protect the trustlet memory regions as well as its own code and data regions from unauthorized access. This approach allows, for example, measured launch procedures [235] to be implemented without additional dedicated hardware support, although some usages may warrant hardware accelerators such as a signature verification engine.

The configured code and data regions are recorded in the Trustlet Table, a write-protected table in on-chip memory, such that they can be looked up and validated by individual trustlets or attestation routines. Only then does the Secure Loader continue to load and execute untrusted software, such as the embedded OS.

Note that external DRAM will not typically be used for confidential trustlet data in this approach. Instead this extended bulk memory can be used to support a larger untrusted OS and applications stack, or public code and data requiring only integrity protection. Moreover, we emphasize that the Secure Loader itself is only active at initialization time. It only configures the hardware memory protection and optionally initializes a chain of trust for remote attestation and trusted execution before delegating control to actual runtime code, such as an untrusted OS.

In the following we discuss the key components of this architecture in more detail, before discussing the initialization and interaction of trustlets in Section 9.4.

9.3.2 Memory Protection Unit (MPU)

To realize flexible and OS-independent memory access control at low cost, we employ a generalized execution-aware MPU design, which also considers the address of the currently executing instruction when validating a particular data or code access.

An MPU can be seen as a lightweight Memory Management Unit (MMU). However, MMUs are primarily designed to implement paging and virtual memory, which in turn can be used to realize access control on physical memory. As such, MMUs impose a large management overhead in form of page tables, which map virtual to physical memory pages and manage their respective access rights. As the page tables typically reside in external, off-chip memory, lookups can incur significant and variable processing delays, which is unacceptable in many real-time application scenarios.

In contrast, an MPU is primarily designed for lightweight access control and does not provide virtual memory. For this purpose, available physical memory is organized into a number of memory protection regions with associated access permissions. The access control rules are not kept in main memory but in local registers available to the MPU. Hence the number of protection regions is determined at production time, e.g., by instantiating an MPU with 12 or 16 such region registers [151, 296, 36].

To support a larger number of protected OS tasks, an MPU is typically combined with CPU privilege levels such as supervisor and user mode execution. In this way, the OS can
program the MPU rules for the next respective task to be scheduled. However, a major drawback of this approach is that the embedded OS and the facilities it implements, such as hardware drivers, communication protocols and other complex abstraction facilities, becomes a single point of failure for platform security enforcement, reducing the resilience of the overall platform to malicious attacks.

**Execution-aware Memory Protection**

Existing MPU implementations enforce execute and read/write access control for different CPU privilege levels on a set of memory regions. For code execute permissions, addresses generated by the CPU’s instruction fetch unit are checked against the programmed access control rules, while for data read/write permissions data addresses generated by the instruction execute unit are monitored.

We enhance this mechanism by providing a means to link code regions to data regions, thus making the permission check *execution-aware*. In addition to validating the data address generated by the instruction execute unit, the instruction address of the executing instruction is also considered. The resulting scheme is illustrated in Figure 9.2, where the MPU not only validates data accesses (*object*, read/write/execute) but additionally considers the currently active instruction pointer (*curr_IP*) as the *subject* performing the access.

Hence, our execution-aware MPU can be programmed to autonomously enforce a fine-grained access control based on individual executing code regions. Figure 9.3 illustrates an example access control matrix that can be programmed and enforced by our MPU, showing three subjects “TL-A”, “TL-B”, “OS” and several memory regions they can be given access to, including their own respective code and data regions as well as the memory-mapped registers of, e.g., the MPU and Timer peripherals. By comparison, a regular MPU can only distinguish between user and supervisor access, and requires the OS to program the correct user-level access rules for the respective next scheduled task.
9.3. THE TRUSTLITE SECURITY ARCHITECTURE

<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
<th>MPU Access Control Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROM / Flash</td>
<td>Trustlet A entry</td>
<td>TL-A (0x00-0A)</td>
</tr>
<tr>
<td>PROM / Flash</td>
<td>Trustlet A code</td>
<td>rx</td>
</tr>
<tr>
<td>PROM / Flash</td>
<td>Trustlet B entry</td>
<td>rx</td>
</tr>
<tr>
<td>PROM / Flash</td>
<td>Trustlet B code</td>
<td>r</td>
</tr>
<tr>
<td>PROM / Flash</td>
<td>OS entry</td>
<td>rx</td>
</tr>
<tr>
<td>PROM / Flash</td>
<td>OS code</td>
<td>r</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>Trustlet A data</td>
<td>rw</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>Trustlet A stack</td>
<td>rw</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>Trustlet B data</td>
<td>-</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>Trustlet B stack</td>
<td>-</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>OS data</td>
<td>-</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>OS stack</td>
<td>-</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>MPU flags</td>
<td>r</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>A/B regions</td>
<td>r</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>OS regions</td>
<td>r</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>Timer period</td>
<td>r</td>
</tr>
<tr>
<td>SRAM / DRAM / Flash</td>
<td>handler(ISR)</td>
<td>r</td>
</tr>
</tbody>
</table>

Figure 9.3: Example memory protection table, defining various memory regions and access permissions for two trustlets A and B as well as an OS.

Memory Protection Faults

When the MPU detects a protection violation, a CPU exception is raised and handled as in regular MPU designs. In particular, the MPU protection fault invalidates the executing instruction and thus also any associated (speculatively allowed) data reads and instruction fetches. The CPU exception engine flushes the pipeline and diverts execution to the designated exception handler, providing the violating instruction address and requested data access as arguments.

9.3.3 Peripherals Access

The typical approach for interacting with platform peripherals is that access is limited to one or more privileged tasks. These tasks then implement a low-level hardware driver and provide a more abstract interface to other applications, regulating access and multiplexing available hardware resources.

TrustLite directly supports this paradigm and extends it to allow any trustlet to exclusively control an arbitrary set of hardware peripherals, without depending on the OS or other privileged code. Due to the flexible access control provided by our execution-aware MPU, trustlet code regions can be associated with a number of data regions. Since all peripheral access is implemented using Memory-Mapped I/O (MMIO), i.e., in the form of read/write access to memory address space, access to individual hardware peripherals is provided by the MPU in the same way as any other memory access. For this purpose, the Secure Loader simply defines the MMIO address space of the respective peripheral as an additional read/write data region of the trustlet. This access is usually exclusive, enabling the trustlet to implement multiplexing and access control as desired.
As an example, consider the access control definitions for the MPU and Timer address regions in Figure 9.3. Using MMIO, the MPU is configured by writing to the appropriate memory regions denoted as “flags” and “regions” of the MPU peripheral. Hence, the MPU can also be configured to deny any further access to its own registers by configuring the appropriate MMIO region as read-only. We use this in the Secure Loader in Section 9.3.5 to prevent modification of the MPU by the OS or other software.

As another example, Figure 9.3 also shows a timer peripheral which can be programmed to call a particular function pointer (handler) after a configurable number of timer ticks (period). Timers are often used by an OS to interrupt the CPU and, e.g., switch to the OS task scheduler. By configuring the read-write access of this peripheral, the device can thus be setup to leverage or disable such an OS scheduler.

Observe that this capability can be applied to a variety of peripherals and usages, enabling trustlets that implement secure user I/O [342] or access cryptographic accelerators. By comparison, the Sancus task model requires that all memory and MMIO accessible for a trustlet are wired in to the same contiguous data region, which is unusual and requires close coordination between hardware design, software development and usage [225].

9.3.4 TrustLite Exception Engine

Previous works [287, 88, 225] on memory protection assumed that trusted applications are non-interruptible, resetting the platform on unexpected errors and operating in a cooperative multitasking environment. However, modern embedded systems often have strong requirements with regard to fault tolerance and responsiveness, and fundamentally require efficient hardware interrupts and software fault handling.

In the following, we propose a modified CPU exception engine that maintains the memory isolation of tasks even in the case of hardware and software exceptions. This allows trustlets to leverage (trusted or untrusted) central scheduling services and exception handlers, without affecting the integrity and confidentiality of their data. In particular, TrustLite instantiations with a secure exception engine allow trustlets to be managed by an untrusted embedded OS, similar to other untrusted OS tasks executing on the platform.

Isolating Exceptions

When handling CPU exceptions such as faults, traps and interrupts, typical computing platforms only perform the minimal tasks of saving the stack and instruction pointer before executing the corresponding (software) exception handler. Depending on the type of exception, such handlers may then only save the particular CPU registers required to perform their operation, and restore them before handing control back to the task.

However, this procedure opens trustlets to information leakage attacks. In a preemptive multitasking environment, trustlets can be interrupted at any time, leaking potentially sensitive information from the CPU registers into the exception handlers and OS. To exclude the OS exception handlers from the TCB of our trustlets, we modify the CPU exception engine to store the stack and instruction pointer, as well as general purpose registers into the protected data region of the interrupted trustlet.
The detailed scheme is illustrated in Figure 9.4. When detecting an exception, existing exception handlers typically store the stack pointer, instruction pointer and CPU flags together with any additional exception information on the OS or other higher-privilege stack. The address of this stack is taken from a well-known location, such as the Task State Segment (TSS) on current x86 CPUs. Depending on the type of exception, the control flow is then redirected to the appropriate Interrupt Service Routine (ISR) as indicated by the Interrupt Descriptor Table (IDT). To provide isolation of trustlets in the face of untrusted ISRs, we modify this scheme as shown in Figure 9.4. In particular, the hardware exception engine first (1) stores the CPU state to the current (task or other) stack \(SP_A\), (2) stores \(SP_A\) to the Trustlet Table, into the row matching the currently executing instruction pointer region, and then clears all general-purpose registers. Only then is the OS stack pointer \(SP_{OS}\) restored in step (3), unless already executing from the OS region, and the regular operation of the exception engine can continue by writing the faulting IP value as well as additional error codes onto the new stack \(SP_{OS}\). The appropriate ISR is called in called step (4).

### Return from Exception

Restarting an interrupted trustlet is performed simply by jumping to its respective entry point and does not require additional hardware logic. However, the trustlet must take care to restore its stack pointer \(SP_A\) as the very first instruction, since that instruction may already be followed by another exception leading the exception engine to store the CPU state to the wrong stack\(^1\).

Note that the ISR and OS can distinguish the source of an exception simply by looking up the faulting IP in the trustlet table, and ensure an appropriate re-entry. Hence this design supports ISRs which directly return to the trustlet as well as approaches where the ISR defers to the task scheduler to potentially launch other trustlets and tasks before continuing the originally interrupted trustlet. Additionally, the reported faulting IP of

---

\(^1\)Since the MPU will typically not be configured to allow such accesses, this misbehavior leads to a memory protection fault, effectively terminating the trustlet.
trustlets can be sanitized to always point to the trustlet’s entry vector to avoid information leakage. Our current analysis shows that the approach also works with nested interrupts, where an ISR may be interrupted by another ISR.

Observe that the Trustlet Table is similar to the Task State Segment (TSS) introduced in the Intel 80386 CPUs, and thus well within the scope of modern low-cost platforms. However, while x86 hardware task switching is designed for hardware acceleration, our Trustlet Table and modified exception handler are fundamental security features. In comparison to the previously proposed asynchronous exit in Intel SGX [211], the entry and exit of individual trustlets is implicitly determined by matching the instruction pointer against the defined EA-MPU regions and does not require new CPU instructions.

9.3.5 Secure Initialization and Reset

In the following we describe the Secure Loader, which performs the secure initialization and configuration updates of trustlets and allows an efficient platform reset.

The first process launched by the CPU upon platform reset is a platform initialization procedure that is typically loaded from a hard-wired location in memory. We realize a Secure Loader by having this initialization procedure protect itself using the MPU’s memory access control. This ensures that the Secure Loader can at most be updated by itself, and not by any other software launched at a later time. The routine then loads one or more trustlets into memory and sets up the respective memory protection rules before launching the main OS.

Figure 9.5 illustrates the detailed interaction with the platform’s Programmable ROM (PROM) and RAM. After performing basic platform initialization such as clearing the MPU access control registers in step (1), step (2) initiates the detection and loading of any trustlets found in PROM. For each trustlet, the Loader parses the trustlet’s metadata and sets up the appropriate memory regions in RAM. The Loader then performs a static initialization of the trustlet, setting up its stack, instruction pointer and rewriting the code to restore its stack from the correct location in the Trustlet Table. Additionally, the global Trustlet Table is populated with the identifier, MPU regions and initial stack pointer of each loaded trustlet (cf. Figure 9.4).

In step (3), the Secure Loader programs the MPU in order to enforce the access control requested by the individual trustlets. For instance, code regions will usually be write-protected and data regions restricted to be read/writable only for the respective trustlet, as illustrated in Figure 9.3. Finally, the MPU regions themselves are locked from further unauthorized access by defining their locations in memory (as well as the MMIO locations of the corresponding MPU registers) as read-only. In this way, trustlets and even the OS are unable to interfere with the protection enforced by the MPU.

The OS may be started in step (4) to drive untrusted platform peripherals and implement any non-critical functionality. An OS can also be made trustlet-aware by inspecting the local Trustlet Table created by the Secure Loader, and registering any identified trustlets similarly to regular tasks managed by the OS. This allows the OS to invoke trustlets by calling into their appropriate entry points or manage the invocation of trustlets using the OS scheduler.

Observe that the initialization code is reliably executed at platform reset, and can
9.3. THE TRUSTLITE SECURITY ARCHITECTURE

Figure 9.5: Bootstrapping Trustlets and OS from PROM.

efficiently reconstruct and re-establish the required memory protection rules. This allows
a more efficient bootstrapping compared to prior solutions such as SMART and Sancus,
which require the hardware to sanitize all volatile memory on platform reset [88, 225].

9.3.6 Important Instantiations

We point out that the presented hardware architecture allows for several different instantiations,
depending on the desired functionality, security level and performance.

Apart from the obvious aspect of scaling the number of MPU regions and thus available
trustlets, designers may decide to hardwire certain MPU regions and memory locations
to provide “hardware trustlets” similar to the ROM-based implementation of SMART.
However, we expect that the secure initialization routine will be typically used to initialize
load-time or “firmware trustlets” with the benefit that devices can be reprogrammed for
different applications or even updated in the field. The secure exception engine can
be included if interruptible “usermode trustlets” are desired, e.g., to enable preemptive
scheduling or for the isolated execution of third party, perhaps not fully trusted, code.

The Secure Loader can be extended to provide Secure Boot, i.e., to validate signatures
of software components before their execution. Alternatively, designers may also use the
Secure Loader to implement a minimal root of trust for measurement and reporting, e.g.,
by instantiating it as a hardware trustlet that also implements measurement and reporting
facilities. In either case the platform may require additional secure key storage and
cryptographic hardware accelerators to meet the performance and security requirements.

Naturally, trustlets may also be deployed without any attestation infrastructure at all,
e.g., to provide an OS-independent remote management service similar to Intel AMT [207].
9.4 Trustlet Invocation and Communication

In the following we describe how trustlets are invoked and how they securely communicate with other trusted or untrusted tasks.

9.4.1 Memory Layout and Entry Vectors

The memory layout of a trustlet is defined by the set of memory regions and associated read, write and execute permissions granted by the MPU. A trustlet is therefore defined by a code region with associated entry vector, as well as one or more data regions which may be accessible to one or more trustlets at the same time and with different access rights. While most of these memory regions are fairly standard and self-explanatory, the entry vector requires a more detailed discussion as it is important for understanding how otherwise isolated trustlets can interact in TrustLite.

At its core, the entry vector is a code section of the overall trustlet code that may also be executed by other tasks or trustlets. It consists of one or more well-defined entry points which enable other tasks to call a particular sub-function of the trustlet, while at the same time assuring that a trustlet’s own code execution (and thus the right to access private sub-functions and sensitive data) is limited to a particular set of entry functions. The entry vector is then defined and enforced as one of the memory regions in the MPU access control table, as illustrated by the “entry” areas for the trustlets A and B in Figure 9.3. In this way, only the trustlet itself can execute its complete program code region, while certain other tasks and trustlets may only execute code from the entry vector. Hence, the entry vector is the external interface of a trustlet and should be programmed with great care to avoid information leakage and other exploitation.

Figure 9.6 shows the two fundamental entry functions of a trustlet, continue() and call(). After initialization by the Secure Loader, continue() is used to continue the regular execution of the trustlet by replacing the current CPU state with previously stored values, as illustrated in the pseudo-code of “Trustlet A” in Figure 9.6. The functions are called simply by jumping to their position in code memory, with possible arguments in CPU registers.

Note that multiple call() entry points can be provided for more efficient Inter-Process Communication (IPC). Moreover, the entry functions of trustlets have similar security and functional requirements as ISRs implemented in typical operating systems. Consequently, we also list the ISRs of the OS as part of its entry vector in Figure 9.6.

9.4.2 Inter-Process Communication

Any isolation-based security system is limited by the need of applications (tasks) to communicate efficiently. In the following we discuss how untrusted tasks can communicate with trustlets while otherwise maintaining memory isolation. We also present a new mechanism for secure communication between trustlets, a local trusted channel protocol that allows efficient communication without trusted security kernels or hypervisors.

Our fundamental requirement for secure IPC without a mutually trusted supervising entity (such as a security kernel) is that trustlets never exit. More specifically, a memory
region assigned to a trustlet at boot time should not be re-used for other purposes until system reset. According to our Secure Loader setup (cf. Section 9.3.5), this assumption holds implicitly as no runtime entity is allowed to reconfigure the MPU\(^2\).

**Unprotected Communication**

![Figure 9.6: OS schedules Trustlet A using untrusted IPC. A then performs a local attestation of Trustlet B and establishes a mutually authenticated channel with B.](image)

Typically, IPC is realized using message queues in the operating system, and tasks are notified by the OS when new messages are available. Messages can also be used to negotiate shared memory regions, allowing tasks to efficiently communicate large amounts of data.

Since any information transferred to or from untrusted tasks is implicitly already accessible to the respective untrusted parties, we can adopt the existing OS facilities to realize unprotected IPC in TrustLite. The main difference in our design is that, to maintain memory isolation between OS and trustlets, we realize signaling and short messages with trustlets in a Remote Procedure Call (RPC) fashion, by jumping to the respective trustlet entry points with arguments in CPU registers.

As illustrated in Figure 9.6, trustlets and OS are equipped with at least one IPC entry point `call(type, msg, sender)` for calling particular handler functions of type `type` to process message `msg`. The handler of the message may simply queue the signal in a message buffer reserved in the trustlet data region, or directly process received messages (synchronous IPC). The caller may also define a pointer `sender` to indicate the task and entry point to which the result of the call should be returned or which should be continued after queuing `msg`.

---

\(^2\) Observe that such an entity may still be introduced, and in this case must have a notion of existing tasks and mediate IPC. However, in this case the trust relationships are similar to those of a microkernel OS, cf. [142, 239].
In addition to direct message transmission via CPU registers, larger messages can also be transmitted indirectly, by referring to their location in memory as part of $\text{msg}$. For this purpose, the trustlet’s meta-data should indicate the size and participating tasks for any desired shared memory regions, such that the Secure Loader can configure appropriate access rules in the MPU. Ideally, the program code of the desired participants should be in adjacent memory regions. In this way, only one code and data region register is needed to provide all authorized tasks with access to the particular memory region, whereas otherwise four or more MPU regions would have to be allocated.

Observe that the ability to realize fine-grained isolation and shared memory is heavily limited by the number of MPU region registers available in the particular platform. However, the nature of the trustlets suggests that the number of information flows per trustlet is fairly limited and predictable, and the considered low-cost application scenarios require only a limited flexibility.

### Communication between Trustlets

Re-using OS facilities for IPC is cost-effective, but exposes exchanged data to manipulation, interception or injection attacks by untrusted components. Moreover, the receiver of a call has no assurance about the sender of an IPC. For secure and mutually authenticated communication between trustlets, we propose a simple handshake protocol to enable local trusted channels.

Our design allows trusted IPC to be initiated with a one round handshake, as illustrated in the communication between trustlet A and B in Figure 9.6. To start communication, the initiator (“Trustlet A”) may first perform a local platform inspection to ensure the correctness of the platform configuration and MPU security policy enforcement for the task or trustlet to be contacted (“Trustlet B”, responder). For this purpose, the initiator uses the responder’s identifier or other meta-data to look it up in the Trustlet Table and determine its memory location and relevant code entry points. Using this information, the initiator may perform an additional sanity check by validating the correct isolation of the responder’s memory regions in the MPU registers. The initiator may also validate a cryptographic hash of the responder’s program code to ensure that its code was not maliciously modified, has the latest patch level etc. This can be done by requiring read access to the responder’s program code for the initiator, or by having a trusted third party such as the Secure Loader measure the trustlet’s code region at load time and store it in a well-known location such as the Trustlet Table. Note that memory reads of the MPU registers, Trustlet Table or program code are secure from manipulation by third parties. Since the MPU does not provide virtual memory, an interruption followed by a remapping of the memory region is not possible.

Once the initiator completes its local attestation of the platform and the peer IPC trustlet, it sends a `syn()` message containing the identifier “A” of the initiator, “B” of the responder and a nonce $N_A$. On receiving the `syn()`, responder B may in turn perform a local attestation of the initiator A and on success respond with a corresponding `ack()`, containing the elements of the initial `syn()` as well as a nonce $N_B$ chosen by the responder.

Since the identity of message receivers is ensured by the use of code entry points and secure exception handling, and since the peers can ensure with local attestation that
their respective IPC receivers will not disclose the nonces, a cryptographic session token
\( t_{A,B} \leftarrow \text{hash}(A,B,N_A,N_B) \) can then be used to authenticate subsequent messages in
either direction.

Similarly to untrusted IPC, trusted IPC can also be used to validate the configuration
of a shared memory region.

9.5 Evaluation

Depending on the requirements of the application scenario, TrustLite may be deployed in
a variety of configurations. In the following, we discuss a minimal and full instantiation
of TrustLite in more detail and compare them against two previously proposed low-cost
trusted computing solutions, SMART [88] and Sancus [225].

9.5.1 Prototype Implementation

We implemented our extensions on the Intel Siskiyou Peak research platform [247]. Siskiyou
Peak is a 32-bit, 5 stage pipeline, single-issue architecture targeted primarily at embedded
applications. The processor is organized as a Harvard architecture with separate buses for
instruction, data and memory-mapped I/O spaces and is fully synthesizable. The target
technology was a Xilinx Virtex-6 FPGA. In particular, we realized execution-aware memory
protection by linking the respective code and data regions provided by the stock MPU
as illustrated in Figure 9.2, using the first four bytes of each code region as its respective
entry vector. Additionally, a 32 bit register storing the secure stack pointer location of
each trustlet is associated with each code region to facilitate secure exception handling.

Our modified exception engine logic required only minimal modifications, since the pro-
cess of pushing the CPU state on the stack is largely similar to the regular exception
engine behavior.

On the software side, we deployed a homegrown OS due to lack of an available open
source embedded OS with support for userspace tasks and memory protection. We fitted
the kernel’s bootstrapping routine to work as a Secure Loader, programming the EA-MPU
and protecting itself from later modifications by the runtime OS code. Currently, we use
a linker script for the GNU C compiler to arrange code and data regions in the memory
image such that they can be recognized and protected by the Secure Loader.

9.5.2 Hardware Extensions Cost

Table 9.1 lists the hardware cost of TrustLite in terms of FPGA registers and LUTs.
In particular, we list the hardware overhead of the full execution-aware MPU extension
in terms of cost per security module and extrapolated base cost with zero supported
protection regions. To enable a better comparison with prior work, we consider a security
module to consist of two associated MPU regions, one for code and one for the data of
the security module, although our implementation also supports multiple code and data
regions per module. We should point out that Sancus extends an open source version
of the 16-bit TI MSP430 microcontroller while our implementation platform is a 32-bit
architecture. In terms of FPGA resource utilization, the unmodified Sancus MSP430
core consumes 998 registers and 2322 LUTs in Xilinx Spartan-6. Our targeted 32-bit core consumes 5528 registers and 14361 LUTs in Xilinx Virtex-6 (this figure also includes a 16550 UART). Both Virtex-6 and Spartan-6 share a 6-input LUT architecture and organize these in ‘slices’ containing 4 LUTs with 8 registers; comparisons between Sancus and TrustLite at the LUT/register or slice level are therefore appropriate.

We now consider the base and per-module cost of the security extensions. TrustLite’s fixed costs are 50% of Sancus while the per module cost is roughly 40% less. It should be pointed out the disparities in hardware costs between TrustLite and Sancus are to some extent a result of differing architectural features. Sancus implements a set of instruction set extensions and instantiates a hardware hash implementation which accounts for the increased base cost. While a hash implementation (hardware or software) is not strictly required by TrustLite, there is ample base cost margin to absorb a hardware hash such as Spongent, which has been shown to consume 22 Spartan-6 slices in a representative implementation [202]. Note also that the TrustLite EA-MPU supports a wider 32-bit address space. Scaling our EA-MPU to support the narrower MSP430 16-bit datapath would roughly result in a further 50% saving in FPGA resources.

At the per-module level a 128-bit MAC key is cached by Sancus resulting in a module storage cost of 128 bits which accounts for a significant portion of the register cost. For usages where performance is not a concern, moving to on-the-fly module key generation would result in a 128 register saving for Sancus, however the hardware cost for TrustLite remains competitive when scaling to a 16-bit data path is taken into account.

To better illustrate the impact of our proposed architecture, Figure 9.7 plots the total hardware cost in FPGA slices\(^3\) as a function of the number of supported security modules, irrespective of the employed underlying core. This is reasonable since our EA-MPU is portable to many other processors, performing essentially the same logic with similar effort, as evidenced by several regular MPUs implementations available today [36, 296, 151]. As can be seen in Figure 9.7, the overhead of Sancus protected software modules rises quickly to twice of the cost of their underlying openMSP430 core, allowing them to fit only 9 protected modules at a design point where TrustLite supports 20.

Table 9.1 also shows that the additional overhead of secure exception handling is very low, with the additional fixed costs even staying within the error margin of the probabilistic FPGA synthesis. Figure 9.7 better illustrates the slightly increased cost

<table>
<thead>
<tr>
<th></th>
<th>TrustLite</th>
<th></th>
<th>Sancus</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Regs</td>
<td>LUTs</td>
<td>Regs</td>
<td>LUTs</td>
</tr>
<tr>
<td>Base Core Size</td>
<td>5528</td>
<td>14361</td>
<td>998</td>
<td>2322</td>
</tr>
<tr>
<td>Extension Base Cost</td>
<td>278</td>
<td>417</td>
<td>586</td>
<td>1138</td>
</tr>
<tr>
<td>Cost per Module</td>
<td>116</td>
<td>182</td>
<td>213</td>
<td>307</td>
</tr>
<tr>
<td>Exceptions Base Cost</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Except. per Module</td>
<td>34</td>
<td>22</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 9.1: FPGA resource utilization of execution-aware memory protection per security module and comparison with Sancus [225].

\(^3\)A slice in Xilinx Spartan-6 or Virtex-6 contains 4 6-input LUTs and 8 registers.
Figure 9.7: Hardware overhead of TrustLite and Sancus in total FPGA slices. Despite supporting a 32-bit address space, TrustLite incurs only about half the hardware overhead of Sancus in both, fixed cost and per module cost.

of adding module protection with secure exceptions. Hence, our approach of managing trustlets in a multitasking environment with central scheduling and software fault handling appears quite practical.

The rather low fixed cost of our extensions also justifies instantiations with only one or two protected modules. In particular, a SMART-like system can be instantiated by merging the security-sensitive SMART code with that of our Secure Loader (which does not require entry points on its own), thus creating a protected attestation and trusted execution service using only a single protected module. With a hardware overhead of only 394 slice registers and 599 slice LUTs, such an instantiation can be very attractive compared to the original SMART instantiation that requires an extra 4kB ROM and does not allow software updates [88].

9.5.3 Runtime Overhead of Memory Protection

As already observed in prior work [225], memory region range checks can be parallelized such that they do not increase memory access time which is in the processor critical path. However, the logic which generates the collective memory access exception logarithmically increases in depth with the number of checked memory regions. We experienced no timing closure problems with up to 32 memory protection regions.

The overhead of initializing trustlets in the Secure Loader is also minimal, requiring only three additional writes to MPU registers for each protection region to define the start, end and permission of that region. Alternatively, the MPU access control rules may also be hard-wired into the MPU to further simplify the instantiation and provide additional security assurances. Note that even such limited instantiations may still be configured to
allow software updates, by declaring the code region of a trustlet as writable to itself or to a separate software update service.

Overall, TrustLite enables multiple alternative instantiations with an overall bootstrapping and memory access overhead that is insignificant or not present at all.

9.5.4 Runtime Overhead of Exception Handling

The only hardware-dictated runtime overhead in TrustLite comes from the (optional) secure exception engine. Since the regular exception engine performs only the minimum work necessary and delegates the OS to store and restore general purpose registers, our modified flow of storing and clearing the complete CPU state before switching into the Interrupt Service Routine (ISR) yields a notable runtime overhead by the exception hardware.

In particular, the unmodified exception engine requires about 21 CPU cycles from recognizing the exception to executing the first ISR instruction, which includes the main work of restoring the OS stack and storing the exception error parameters together with the interrupted execution state onto that stack. On top of this flow, our secure exception engine flow takes another 2 cycles to recognize that a trustlet is being interrupted, 10 cycles to store all but the ESP registers onto the trustlet (instead of OS) stack, and 9 cycles to clear all general purpose registers and store the ESP into the Trustlet Table (cf. Section 9.3.4). Overall, our secure exception handler thus incurs a runtime overhead of 21 cycles or 100% of the regular exception flow when interrupting a trustlet, and 2 cycles otherwise.

Considering that a 32-bit i486 CPU takes at least 107 cycles for context switching and larger processors typically require significantly more, we believe this overhead is reasonable [143]. Furthermore, observe that task interruption is often followed by the invocation of the OS scheduler. For our hardware-protected trustlets, the software would be working to manage already cleared CPU registers, thus wasting CPU cycles in the time critical ISR and scheduler code. As part of our future work, we therefore plan to investigate how ISR and OS optimizations can further reduce the average overhead of trustlet exceptions.

9.6 Security Considerations

In the following we informally argue how TrustLite meets each of the system and security requirements in Section 9.2.3.

Data Isolation. As a low-end platform without multiprocessing and virtual memory, TrustLite achieves data isolation simply by memory access control. Hardware prevents unauthorized software from accessing trustlet code and data, restricting it to the explicitly provided code entry points of the respective trustlet. Our secure exception engine extends this separation into the CPU register files, ensuring that no data is unintentionally leaked on context switch. Like all known memory protection schemes, the approach is vulnerable to software exploitation attacks which may subvert trustlet code to leak information [317].
However, while TrustLite assists in reducing the vulnerability of trustlets by minimizing their overall software complexity (TCB), advanced software protection is outside the scope of this work and we instead refer to our assumption in Section 9.2.2 that trustlet code is correct.

**Attestation.** Building on a physical memory address space with full isolation of trustlets from other software, TrustLite ensures that the OS or other software cannot manipulate the outcome of memory read accesses of the MPU register set or other trustlet’s code regions. Hence, any software with the required read permissions can inspect relevant system state and perform actions depending on the outcome of this inspection, without other software being able to interfere.

**Trusted IPC.** Since TrustLite establishes trustlets at platform initialization time and ensures their protection until system reset, a single inspection and validation of another trustlet’s code and meta-data is sufficient to establish its identity, integrity and entry points for IPC. Moreover, since IPC with other tasks consists essentially of a jump instruction into their respective code entry region, the receiver identity and message confidentiality are directly enforced by the CPU. Trusted IPC, consisting of a confidential and mutually authenticated local channel with integrity verification of either endpoint can therefore be established using a single round-trip protocol, as described in Section 9.4.2.

**Secure Peripherals.** The unique memory isolation architecture of TrustLite enables a simple and direct extension to secure peripheral access based on Memory-Mapped I/O (MMIO). This enables the construction of secure device drivers and extends trusted software execution to assure interaction with device users or other aspects of a platform’s environment. For future work, we want to extend this secure interaction to (possibly untrusted) devices with Direct Memory Access (DMA) capability, which were shown to be problematic for certain security architectures [240].

**Fast Startup.** While prior works [88, 225] require the hardware to purge all volatile memory on platform reset, our Secure Loader allows secure re-initialization of the memory protection rules before invoking untrusted software and only needs to clear data regions that should be re-allocated to other software. Moreover, TrustLite allows the measurement of trustlet code regions on demand, reducing the platform initialization overhead.

**Protected State.** Similar to other more recent approaches to trusted execution [211, 225], TrustLite allows trustlets to maintain a transient execution state across invocations. This significantly reduces the activation latency of trustlets, resulting in reduced power consumption and better application performance and user experience.

**Field Updates.** TrustLite security extensions are independent of the CPU instruction set and completely programmable by software. This enables updates to any trusted or untrusted software, security policy and potentially also the Secure Loader itself.
Fault Tolerance. Our secure exception engine and secure peripherals enable TrustLite instantiations that tolerate software faults in trustlets or even in the OS. In particular, TrustLite trustlets can cooperate with an untrusted OS but may also implement ISRs and hardware drivers on their own, thus preventing trivial denial-of-service attacks that affect several prior works [290, 210, 88, 225].

Overall, TrustLite achieves all security and functional requirements while being comparable or better than previously proposed solutions.

9.7 Related Work

Memory protection is perhaps the oldest hardware security mechanism in computing. Currently deployed systems typically use an MMU or MPU to prevent less privileged software from manipulating foreign memory. Alternatively, the Intel 80286 introduced memory segmentation where the hardware associates segments with a two bit privilege level and prevents access by software unless it is running at least at the same privilege level. Execution-Aware Memory Protection is fundamentally different from these concepts as it is not based on privilege levels but directly associates code and data regions in memory. This allows us to support a large number of software modules which are isolated independently of the OS or another trusted software runtime.

CPU Extensions. Several works also proposed CPU extensions to protect the execution of software. Aegis [290] loads code from untrusted memory and performs integrity verification and memory encryption in the CPU. Alternatively, it was proposed to extend the code of trusted software modules with integrity check values that are automatically verified by the CPU once loaded into its cache registers [97]. Intel Trusted Execution Technology (TXT) [134] and AMD Pacifica [24] introduce a secure re-initialization of the CPU at runtime. Intel SGX [211] provides hardware-protected execution of multiple concurrent userspace applications. However, these existing works focus on PCs, rich mobile platforms and servers, and it is unclear if they can be economically deployed to low-cost systems.

Secure Hypervisors and Kernels. A large amount of research considers how to extend and improve software protection based on newly introduced privilege-based protection facilities [220, 35]. Higher privilege modes can be used by a secure hypervisor or OS kernel to provide strong isolation of certain critical software modules [216, 142, 178, 239, 208, 206], assuming that sufficient attention is paid to reduce the complexity of the privileged secure OS or hypervisor [60, 145, 342]. We believe that a secure and minimal OS kernel can be of great benefit for increasing the resilience of TrustLite systems, while at the same time TrustLite can be useful for security kernels to simplify and automate software isolation.

Software-based Attestation. Software-based attestation exploits the computational limitations of a platform to ensure that only a particular algorithm can be executed within a given time frame. The scheme has been applied to a variety of platforms and use-cases,
including sensor networks, voting machines and trusted key deployment [271, 192, 113, 185]. Since purely software-based attestation must assume any stored keys to be compromised, several works also consider the combination of software-based attestation with hardware trust anchors [185, 287, 160, 257, 171]. However, even if assuming a minimal hardware trust anchor, the scheme requires a temporary full utilization of the platform’s computational capacity, and is unable to sustain more than one trusted execution environment. Hence, software-based attestation appears more suitable for attestation of legacy devices or initial trust establishment for platforms without trusted attestation keys and certificates.

**SMART.** SMART [88] enables remote attestation and trusted execution on low-cost systems using a custom access control on the memory bus. Specifically, they only allow access to a particular secret key in memory if the current CPU instruction pointer points to a trusted code region in ROM. The instruction pointer in turn may only point into this trusted code if it previously also pointed into the same region or if it currently points to the very beginning of that code. As a result, the secret key is only accessible by the trusted code in ROM, and can be used to prove to other parties that the ROM code has been executed correctly, e.g., has performed a measurement of some local software.

While our memory access control model is inspired by SMART, we extend and generalize it in several ways. Instead of deploying an attestation and trusted execution routine in ROM with custom instruction pointer restrictions, TrustLite realizes a generic, programmable access control logic, enabling a large variety of software programs that can be updated in the field. Moreover, TrustLite supports concurrent execution of trusted applications whereas SMART requires applications to store and restore their state on each invocation, resulting in significant overhead.

**Sancus and SPM.** Software-Protected Module (SPM) [287] proposes new CPU instructions that allow tasks to be measured and loaded into protected memory regions and to query the status of other task’s memory protection. Assuming publicly readable program code segments, protected tasks can then inspect and attest each other in physical memory. Sancus [225] implements and extends SPM for the openMSP430 CPU. They store measurements of loaded tasks in protected registers and provide special instructions to authenticate and accelerate task IPC.

Similarly to TrustLite, SPM enforces certain code entry points for communication and multi-tasking. However, IPC and inspection of other tasks is simpler in TrustLite since our EA-MPU protection rules persist until platform reset. Furthermore, our more generic handling of memory protection regions allows trustlets to have multiple code and/or data regions, enabling secure peripheral access and simple shared memory constructions. In contrast to Sancus and SPM, TrustLite also supports the secure interruption of trusted tasks, and our Secure Loader solves the problem of sanitizing memory after platform reset by simply restoring the required access protection rules.
9.8 Limitations

We acknowledge that TrustLite also suffers from certain limitations. Notably, the limited number of supported memory protection regions in the MPU and the fact that trustlet memory regions cannot be re-allocated without a platform reset. Considering the typically very targeted and tight requirements on low-end embedded systems, we believe that these limitations are reasonable as they allow a simplified (low-cost) design of the memory protection and IPC subsystems. In particular, MPUs are well established in embedded systems and the software stack of such systems is usually very carefully adapted for the particular use case. Since most of the security mechanisms in TrustLite are programmable in software, a vendor may also deliberately deploy a hardware platform that does not support all possible usages simultaneously but instead detects the desired scenario and establishes the required software stack and protection facilities in a second boot phase during deployment.

9.9 Summary

We have presented TrustLite, a new security architecture for providing trusted computing functionality on low-cost embedded systems. Our design represents a new alternative for isolating secure applications, providing trusted execution, OS interoperability and secure peripheral access.

Our architecture is enabled by a generalized memory protection scheme, combining an execution-aware memory protection unit (EA-MPU) with a secure exception engine that protects the task state from untrusted exception handlers. Our Secure Loader provides a simple yet flexible establishment and update of the platform’s security policy, prevents memory leakage after platform reset and can be extended to act as a root of trust for remote attestation and trusted execution. Depending on the application scenario and cost constraints, TrustLite can be instantiated in several configurations, from providing a single atomic firmware security service to isolating userspace tasks in a preemptive multi-tasking environment.

For future work, we want to investigate the integration of cryptographic accelerators with TrustLite and evaluate its impact on IPC performance and context switching.
Chapter 10

Conclusion and Future Work

Trusted Computing has come a long way. Initial hopes and fears of having an absolute and complete view into the state of remote computer systems have been disappointed by the enormous complexity and indeterminacy of today's software systems.

However, current products increasingly deploy some first, more conservative security techniques such as a TPM-based secure key storage and hard disk encryption. On the mobile devices front, significant effort is being invested to achieve flexible secure credential management using trusted execution. In general, the more limited and controlled software environment of mobile and low-cost embedded devices reduces the complexity of measuring and assessing the software state of a system. This is particularly true for the emerging Trusted Execution Environments (TEEs), which execute small amounts of software logic isolated from the remaining system.

In this thesis, we have investigated the problem of trusted communication and trust establishment in distributed embedded systems. In the first part, we have explored the potential and limitations of trusted virtual networks. We have designed a secure IPsec-based VPN in Chapter 3 that exchanges remote attestation messages in a flexible and efficient way. We then investigated how the often neglected problem of covert channels affects the security of such virtual networks. We explored the scenario of strong inside adversaries and covert channel leakage in Chapter 4 and a scenario with a weaker outside adversary performing traffic analysis in Chapter 5. Our work shows that covert channel and traffic analysis attacks can be mitigated at reasonable costs, especially if possible trade-offs and combinations of security mechanisms are considered.

In the second part we investigated the problem of establishing trust in low-end devices with often severely constrained computational performance, memory or power supply. We performed a systematic analysis of software-based attestation in Chapter 6, formalizing required assumptions and pointing out several unsolved problems. In Chapter 7 we propose a combination of purely software-based attestation with Physically Unclonable Functions (PUFs), yielding a new remote attestation protocol that binds the attestation to particular devices and requires only minimal trusted hardware logic.

Our use-case study on Software Smartcards in Chapter 8 showed that TEEs can yield significant security and usability improvements as they enable a flexible yet secure compromise between complex applications and small and reliable program logic for the security-critical parts of a program. Finally, we presented TrustLite, a new approach to realize
TEEs even on low-cost, resource-constrained devices in Chapter 9. TrustLite leverages the reduced hardware complexity of low-end platforms to realize a low-cost, hardware-enforced, and OS-independent isolation of tasks, which can scale from isolating small cryptographic services to hardware drivers and operating systems.

Overall, we have explored and demonstrated the potential to scale trusted computing technology even to low-end, resource constrained devices, thus enabling automated, distributed security assessment and enforcement and thus a significantly increased resilience of complex distributed embedded systems to attacks.

For future work, a major challenge remains to develop and adopt the higher layer trust management schemes and protocols which are required to collect and judge the status information of available devices in a distributed system, and to automatically derive, direct and monitor appropriate responses. A major practical challenge in this context remains the problem of verifying a remote attestation report, in particular the problem of building appropriate reference databases and identifying relevant measurements.

Furthermore, despite the recent emergence of “cloud” computing and several associated research projects, the deployment and operation of trustworthy virtual infrastructures on top of untrusted computing infrastructures remains an open problem. However, modern IT infrastructures can only be secured if the server or cloud backends can be trusted.

One of our own major goals is the further extension, prototyping and evaluation of the TrustLite architecture. For instance, significant advancements have been made in safe programming languages and the automated, formally verifiable synthesis of low-level software, which may be suitable to further increase the assurance in the system TCB.
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADP</td>
<td>Attestation Data Payload</td>
<td>32</td>
</tr>
<tr>
<td>AH</td>
<td>Authenticated Header</td>
<td>19</td>
</tr>
<tr>
<td>AIK</td>
<td>Attestation Identity Key</td>
<td>10</td>
</tr>
<tr>
<td>AK</td>
<td>Attestation Key</td>
<td>34</td>
</tr>
<tr>
<td>APN</td>
<td>Access Point Name</td>
<td>79</td>
</tr>
<tr>
<td>AS</td>
<td>Attestation Service</td>
<td>33</td>
</tr>
<tr>
<td>BIOS</td>
<td>Basic Input/Output System</td>
<td>11</td>
</tr>
<tr>
<td>BPMT</td>
<td>Block-based Pseudorandom Memory Traversal</td>
<td>114</td>
</tr>
<tr>
<td>BSC</td>
<td>Base Station Controller</td>
<td>73</td>
</tr>
<tr>
<td>BTS</td>
<td>Base Transceiver Station</td>
<td>73</td>
</tr>
<tr>
<td>CE</td>
<td>Computing Engine</td>
<td>93</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>11</td>
</tr>
<tr>
<td>CRP</td>
<td>Challenge Response Pair</td>
<td>130</td>
</tr>
<tr>
<td>CRTM</td>
<td>Core Root of Trust for Measurement</td>
<td>11</td>
</tr>
<tr>
<td>CSP</td>
<td>Cryptographic Service Provider</td>
<td>140</td>
</tr>
<tr>
<td>DAA</td>
<td>Direct Anonymous Attestation</td>
<td>10</td>
</tr>
<tr>
<td>DF</td>
<td>Don’t Fragment</td>
<td>52</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
<td>169</td>
</tr>
<tr>
<td>DoS</td>
<td>Denial of Service</td>
<td>33</td>
</tr>
<tr>
<td>DPI</td>
<td>Deep Packet Inspection</td>
<td>78</td>
</tr>
<tr>
<td>DRM</td>
<td>Digital Rights Management</td>
<td>16</td>
</tr>
<tr>
<td>DRTM</td>
<td>Dynamic Root of Trust for Measurement</td>
<td>122</td>
</tr>
<tr>
<td>DS</td>
<td>Differentiated Services</td>
<td>75</td>
</tr>
<tr>
<td>DTLS</td>
<td>Datagram TLS</td>
<td>20</td>
</tr>
<tr>
<td>EAP</td>
<td>Extensible Authentication Protocol</td>
<td>37</td>
</tr>
<tr>
<td>ECN</td>
<td>Explicit Congestion Notification</td>
<td>75</td>
</tr>
<tr>
<td>EK</td>
<td>Endorsement Key</td>
<td>10</td>
</tr>
<tr>
<td>EMSCB</td>
<td>European Multilaterally Secure Computing Base</td>
<td>27</td>
</tr>
<tr>
<td>ESP</td>
<td>Encapsulated Security Payload</td>
<td>64</td>
</tr>
<tr>
<td>Flags</td>
<td>IPv4 Flags</td>
<td>52</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
<td>8</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>GGSN</td>
<td>Gateway GPRS Support Node</td>
<td>73</td>
</tr>
<tr>
<td>GPRS</td>
<td>General Packet Radio Service</td>
<td>73</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
<td>28</td>
</tr>
<tr>
<td>HPCM</td>
<td>High-Performance Covert Channel Mitigation</td>
<td>62</td>
</tr>
<tr>
<td>HPET</td>
<td>High-Precision Event Timer</td>
<td>62</td>
</tr>
<tr>
<td>ICE</td>
<td>Indisputable Code Execution</td>
<td>114</td>
</tr>
<tr>
<td>ICMP</td>
<td>Internet Control Message Protocol</td>
<td>54</td>
</tr>
<tr>
<td>IDT</td>
<td>Interrupt Descriptor Table</td>
<td>159</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
<td>19</td>
</tr>
<tr>
<td>IKE</td>
<td>Internet Key Exchange</td>
<td>19</td>
</tr>
<tr>
<td>IKEv2</td>
<td>Internet Key Exchange version 2</td>
<td>28</td>
</tr>
<tr>
<td>IMA</td>
<td>Integrity Measurement Architecture</td>
<td>12</td>
</tr>
<tr>
<td>IMAP</td>
<td>Internet Message Access Protocol</td>
<td>86</td>
</tr>
<tr>
<td>IM</td>
<td>Instant Messaging</td>
<td>78</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
<td>162</td>
</tr>
<tr>
<td>IPD</td>
<td>Inter-Packet Delay</td>
<td>56</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
<td>18</td>
</tr>
<tr>
<td>ISN</td>
<td>Initial Sequence Number</td>
<td>82</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
<td>159</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
<td>48</td>
</tr>
<tr>
<td>MF</td>
<td>More Fragments</td>
<td>52</td>
</tr>
<tr>
<td>MITM</td>
<td>Man-in-the-Middle</td>
<td>64</td>
</tr>
<tr>
<td>MMIO</td>
<td>Memory-Mapped I/O</td>
<td>152</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
<td>155</td>
</tr>
<tr>
<td>MPU</td>
<td>Memory Protection Unit</td>
<td>155</td>
</tr>
<tr>
<td>MSS</td>
<td>Message Segment Size</td>
<td>83</td>
</tr>
<tr>
<td>MTM</td>
<td>Mobile Trusted Module</td>
<td>11</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum Transmission Unit</td>
<td>76</td>
</tr>
<tr>
<td>NAPT</td>
<td>Network Address Port Translation</td>
<td>83</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
<td>75</td>
</tr>
<tr>
<td>NGSCB</td>
<td>Next Generation Secure Computing Base</td>
<td>27</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
<td>83</td>
</tr>
<tr>
<td>ObC</td>
<td>On-board Credentials</td>
<td>149</td>
</tr>
<tr>
<td>OpenTC</td>
<td>Open Trusted Computing</td>
<td>27</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
<td>11</td>
</tr>
<tr>
<td>PAL</td>
<td>Piece of Application Logic</td>
<td>13</td>
</tr>
<tr>
<td>PBA</td>
<td>Property-based Attestation</td>
<td>17</td>
</tr>
<tr>
<td>PCR</td>
<td>Platform Configuration Register</td>
<td>10</td>
</tr>
<tr>
<td>PKCS</td>
<td>Public Key Cryptography Standard</td>
<td>8</td>
</tr>
</tbody>
</table>
LIST OF ACRONYMS

PKI  Public Key Infrastructure .................................................. 35
PM   Primary Memory ............................................................. 93
PMTUD Path MTU Discovery ...................................................... 55
PMTU Path MTU ........................................................................ 77
PRF  Pseudo-Random Function .................................................. 124
PRIMA Policy-reduced IMA ......................................................... 12
PRNG Pseudo-Random Number Generator ................................... 126
PROM Programmable ROM ....................................................... 152
PropMgr Property Manager ....................................................... 29
PUF  Physically Unclonable Function .......................................... 173
RBAC Role-Based Access Control ............................................... 42
RED Random Early Detection .................................................. 61
RPC  Remote Procedure Call .................................................... 163
RTT  Round-Trip Time ............................................................ 83
SACK Selective ACK ............................................................... 83
SA   Security Association ........................................................ 20
SCUBA Secure Code Update By Attestation ................................ 128
SETA Secure Execution Token Application ................................ 141
SGSN Serving GPRS Support Node ............................................. 73
SIM  Subscriber Identity Module ................................................ 18
SML  Stored Measurement Log .................................................. 10
SMM  System Management Mode ............................................... 13
SM   Secondary Memory .......................................................... 93
SMTP Simple Mail Transfer Protocol ........................................ 86
SoC  System on Chip ............................................................... 152
SPI  Security Parameter Index ................................................... 64
SPM  Software-Protected Module ............................................... 171
SRAM Static RAM ................................................................. 110
SRK  Storage Root Key ........................................................... 29
sVPN Secure VPN ................................................................. 27
TCB  Trusted Computing Base .................................................. 152
TCG  Trusted Computing Group ............................................... 34
TCP  Transmission Control Protocol ......................................... 20
TEE  Trusted Execution Environment .......................................... 173
TFC  Traffic Flow Confidentiality ............................................... 62
TLS  Transport Layer Security .................................................. 38
TNCCS TNC Client Server ...................................................... 32
TNC  Trusted Network Connect ................................................ 28
TOCTOU Time Of Check to Time Of Use .................................... 96

LIST OF ACRONYMS
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM</td>
<td>Trusted Platform Module</td>
<td>123</td>
</tr>
<tr>
<td>TSS</td>
<td>Task State Segment</td>
<td>159</td>
</tr>
<tr>
<td>TS</td>
<td>Timestamp</td>
<td>83</td>
</tr>
<tr>
<td>TTL</td>
<td>Time To Live</td>
<td>75</td>
</tr>
<tr>
<td>TVD</td>
<td>Trusted Virtual Domain</td>
<td>27</td>
</tr>
<tr>
<td>TXT</td>
<td>Trusted Execution Technology</td>
<td>170</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
<td>20</td>
</tr>
<tr>
<td>VDC</td>
<td>Virtual Data Center</td>
<td>40</td>
</tr>
<tr>
<td>VLAN</td>
<td>Virtual LAN</td>
<td>42</td>
</tr>
<tr>
<td>VMM</td>
<td>Virtual Machine Monitor</td>
<td>39</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
<td>28</td>
</tr>
<tr>
<td>VoIP</td>
<td>Voice over IP</td>
<td>5</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
<td>68</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
<td>48</td>
</tr>
<tr>
<td>Wscale</td>
<td>Window Scale</td>
<td>83</td>
</tr>
</tbody>
</table>
List of Publications


Bibliography


[77] B. X. Chen. AT&T tells free tethering customers it’s time to pay up. Wired, 2011.


[132] K. Goldman, R. Perez, and R. Sailer. Linking remote attestation to secure tunnel endpoints. In Juels et al. [166].


[149] P. Horowitz. AT&T cracking down on unofficial iPhone tethering & MyWi users. OSXDaily, 2011.


BIBLIOGRAPHY


[266] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In Juels et al. [165].


