12-2010

Active Bundles for Protecting Confidentiality of Sensitive Data Throughout Their Lifecycle

Lotfi Ben Othmane
Western Michigan University

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation
Othmane, Lotfi Ben, 'Active Bundles for Protecting Confidentiality of Sensitive Data Throughout Their Lifecycle' (2010).
Dissertations. 616.
https://scholarworks.wmich.edu/dissertations/616

This Dissertation-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Dissertations by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
ACTIVE BUNDLES FOR PROTECTING CONFIDENTIALITY OF SENSITIVE DATA THROUGHOUT THEIR LIFECYCLE

by
Lotfi Ben Othmane

A Dissertation
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Doctor of Philosophy
Department of Computer Science
Advisor: Leszek Lilien, Ph.D.

Western Michigan University
Kalamazoo, Michigan
December 2010
ACTIVE BUNDLES FOR PROTECTING CONFIDENTIALITY OF SENSITIVE DATA THROUGHOUT THEIR LIFECYCLE

Lotfi Ben Othmane, Ph.D.
Western Michigan University, 2010

Protecting confidentiality of shared sensitive data requires satisfying conflicting needs for disseminating data and preventing unauthorized data disclosures. We propose a solution named the active bundles scheme for protecting sensitive data from their disclosures to unauthorized parties during their dissemination. The scheme protects data throughout their entire lifecycle, from data creation through their dissemination to their evaporation or apoptosis (a partial or complete self-destruction, respectively).

An active bundle packages together sensitive data, metadata, and a virtual machine (VM) specific to the bundle. Metadata contain information related to the use of data, including data access control and dissemination policies. A VM controls all activities of its active bundle, and enforces the policies specified by metadata. Implementing VMs in effective and efficient ways is the key issue for the scheme.

There are seven main contributions of this Thesis. First, we propose the active bundles scheme. Second, we identify and investigate four different VM implementations: (i) using trusted third parties (TTPs), (ii) utilizing mobile agents and their frameworks, (iii) using autonomous applications based on secure computing, and (iv) using autonomous applications based on obfuscated control flow graphs. Third, we show that there are no
available solutions for protecting confidentiality of code and data carried by mobile agents providing output to visited hosts. Fourth, we build a TTP-based prototype of the active bundle scheme, which demonstrates the practicality of the scheme. Fifth, we prove that there is no universal privacy-homomorphic decryption function, and there exists no universal secure autonomous sequential VM for an encrypted decryption function. Sixth, we pioneer the use of secure computing for program obfuscation. Seventh, we present a sample application of active bundles for identity management in cloud computing.

We believe that these contributions justify our thesis: *Data can protect themselves from unauthorized accesses by malicious hosts.* This is possible due to two salient features of the active bundle scheme: making data inseparable from associated metadata and VMs, and making data active; that is, able to protect themselves from unauthorized disclosures.
Dedicated to my daughters Mariam, Shaima, and Zaineb.
I am grateful to Allah for giving me the endurance to do this work.

This Thesis was for me a journey. During this journey, I did not only gain deep knowledge about the research subject, but also learned about collaborating with colleagues, and writing research results in a convincing manner. This wouldn’t have been possible without the support and guidance of many friends with whom I worked during this journey.

First, I would like to express my most sincere appreciation and profound gratitude to my advisor Professor Lilien, for his guidance and support during all the stages of this Thesis. I am honored to have worked with Prof. Lilien. I appreciate his dedication and tireless efforts to help me learn and succeed. I value the opportunity to learn from him not only his insights in the research subject, but also the taste of discovering new ideas and the rigor in reporting the results of testing these ideas. In short, he is inspiring to me. I also thank him for introducing me to his collaborators.

I would like to extend my sincere gratitude to Professors Bharat Bhargava from Purdue University who continuously shared his insights in the research subject and provided me with his critique to the idea. I thank him for introducing me to his collaborators and his support during the stages of the research.
I greatly value the support of my Thesis committee members: Professors Ala Al-Fuqaha, Ajay Gupta, and Zijiang (James) Yang. I would like to thank you all for your valuable time, discussions, comments, and suggestions during all stages of my work on this Thesis.

I am also so grateful to the faculty and staff of the Department of Computer Science for helping me during this journey. My special thanks go to the Department Chair, Prof. Donald Nelson; the faculty: Professors Donna Kaminski, Elise de Doncker, Mark Kerstetter, Dionysios Kountanis, and Ron Miller; and the department staff Natallie Bolliger, John Horton, and Sheryl Todd.

I would like also to thank my colleagues from WMU and Purdue with whom I collaborated in several works during this quest: Pelin Angin, Mehdi Azarmi, David Tannor, Rohit Ranchal, Raed M. Salih, and Nupur Singh.

I would like to thank my father Hedi, my mother Salha, and my brothers and sisters Nejib, Nizar, Sami, Awatef, and Nawel for their moral and financial support. I would like to thank my wife, Nejia, for her help during this odyssey. I also would like to thank my daughters Mariam, Shaima, and Zaineb for their sacrifices.
I would like also to thank my friends for their help and support during this journey: Bilal Abu Bakr, Adawia Al-Alawneh, Nabeel Alshabtat, Osama Awaad, Mohamed Elbes, Mohamed ElWakil, Nishant Gupta, Mahmoud Said, Abdullah Shehabat, Stephany Coffman-Wolph, Jing Zhang, and Imad Zyout.

Lotfi Ben Othmane
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGMENTS</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>xv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xvi</td>
</tr>
</tbody>
</table>

## CHAPTER

1. INTRODUCTION
   1.1. Definitions of Basic Terms
   1.2. Motivation for Protecting Confidentiality of Sensitive Data
   1.3. Problem Statement and Goal
   1.4. The Proposed Approach and Its Overview
   1.5. Main Thesis Contributions
   1.6. Thesis Organization

2. LITERATURE REVIEW ON CONFIDENTIALITY OF DISSEMINATED SENSITIVE DATA
   2.1. Introduction
   2.2. Confidentiality Protection Solutions for Enumerable Receiving Entities
   2.2.1. Protecting Confidentiality of Sensitive Data Using Multiple Keys
   2.2.2. Protecting Confidentiality of Sensitive Data Using Policies and TTPs
   2.2.3. Protecting Confidentiality of Sensitive Data Using Trusted VMs
<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.</td>
<td>Confidentiality Protection Solutions for Non-Enumerable Receiving Entities</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1.</td>
<td>Protecting Confidentiality of Sensitive Data Using Sticky Policies and Identity-based Encryption</td>
<td>18</td>
</tr>
<tr>
<td>2.3.2.</td>
<td>Protecting Confidentiality of Sensitive Data Using the Vanish System</td>
<td>20</td>
</tr>
<tr>
<td>2.3.3.</td>
<td>Protecting Confidentiality of Sensitive Data Using Data Tethers</td>
<td>22</td>
</tr>
<tr>
<td>2.3.4.</td>
<td>Protecting Confidentiality of Sensitive Data Using Activation Time</td>
<td>22</td>
</tr>
<tr>
<td>2.3.5.</td>
<td>Protecting Confidentiality of Sensitive Data Using Bundles and Active Bundles</td>
<td>23</td>
</tr>
<tr>
<td>2.4.</td>
<td>Conclusions</td>
<td>24</td>
</tr>
<tr>
<td>3.</td>
<td>INTRODUCTION TO SECURE COMPUTING</td>
<td>25</td>
</tr>
<tr>
<td>3.1.</td>
<td>Introduction</td>
<td>25</td>
</tr>
<tr>
<td>3.2.</td>
<td>Three Secure Computing Approaches</td>
<td>26</td>
</tr>
<tr>
<td>3.2.1.</td>
<td>Protecting Data Using Secure Computing with Encrypted Data (CED)</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2.</td>
<td>Protecting Functions Using Secure Computing with Encrypted Functions (CEF)</td>
<td>28</td>
</tr>
<tr>
<td>3.2.3.</td>
<td>Protecting Data and Functions Using Secure Computing with Encrypted Data and Encrypted Functions (CEDEF)</td>
<td>30</td>
</tr>
<tr>
<td>3.3.</td>
<td>Methods for Secure Computing</td>
<td>32</td>
</tr>
</tbody>
</table>
CHAPTER

3.3.1. Coding Theory ................................................................. 34
3.3.2. Garbled Circuits ............................................................... 34
3.3.3. Hardware-based Methods ............................................... 38
3.3.4. Obfuscation ................................................................. 38
3.3.5. Privacy Homomorphisms .............................................. 39
3.3.6. TTP-based Methods .................................................... 43
3.3.7. Threshold-based Methods ............................................. 43
3.3.8. Function Composition .................................................. 46
3.3.9. Hybrid Methods ........................................................... 46
3.3.10. Other Methods .......................................................... 47
3.4. Summary ............................................................................. 47

4. ACTIVE BUNDLE SCHEME ...................................................... 49
4.1. Introduction ......................................................................... 49
4.2. Related Approaches for Protecting Confidentiality of Disseminated Sensitive Data ...................................................... 49
4.2.1. Using a Trusted Platform Module (TPM) ...................... 50
4.2.2. Using a Privacy Policy-based Identity Management Encryption Scheme ......................................................... 50
4.2.3. Using an Algorithm-embedded Context-aware Policy ...... 51
4.3. Our Approach: Protecting Sensitive Data Using Active Bundles ................................................................. 51
Table of Contents—Continued

CHAPTER

4.4. Structure and Operation of Active Bundles .............................. 54
  4.4.1. Structure of Active Bundles ................................................ 54
  4.4.2. Operation of Active Bundles .............................................. 56
4.5. Summary ................................................................................... 57

5. PROTECTING CONFIDENTIALITY OF MOBILE AGENTS
  PROVIDING OUTPUT TO VISITED HOSTS ................................. 59
  5.1. Introduction ............................................................................... 59
     5.1.1. Mobile Agents, Their Uses, and Advantages...................... 60
     5.1.2. Motivation of Investigating Confidentiality of Mobile
            Agents ................................................................................. 65
     5.1.3. Analysis of Mobile Agents’ Confidentiality ....................... 65
     5.1.4. Desirable and Confidentiality-related Properties of Mobile
            Agents ................................................................................. 71
     5.1.5. Chapter Goal and Summary of Assumptions ...................... 74
     5.1.6. Chapter Organization .......................................................... 77
     5.2. Threat Model for Mobile Agents Providing Output to Visited
          Malicious Hosts ........................................................................ 77
     5.3. Applicability of Secure Computing Approaches for Protecting
          Mobile Agents ............................................................................. 79
          5.3.1. Applicability of Secure Computing with Encrypted Data
                 (CED) for Protecting Mobile Agents .................................. 79
          5.3.2. Applicability of Secure Computing with Encrypted
                 Functions (CEF) for Protecting Mobile Agents ................. 80
CHAPTER

5.3.3. Applicability of Secure Computing with Encrypted Data and Encrypted Functions (CEDEF) for Protecting Mobile Agents ................................................................................ 81

5.4. Applicability of Secure Computing Methods to Protecting Mobile Agents .......................................................................................... 83

5.4.1. Coding Theory ........................................................................ 83

5.4.2. Garbled Circuits .................................................................... 83

5.4.3. Hardware-based Methods ................................................... 84

5.4.4. Obfuscation ......................................................................... 84

5.4.5. Privacy Homomorphisms.................................................... 85

5.4.6. TTP-based Methods ............................................................ 86

5.4.7. Threshold-based Methods .................................................. 86

5.4.8. Function Composition......................................................... 86

5.4.9. Hybrid Methods .................................................................. 87

5.4.10. Other Methods ................................................................... 87

5.5. Nine Available Solutions for Secure Computing Protecting Mobile Agents Providing Output to Visited Malicious Hosts ........................................................................ 89

5.5.1. Solution S1: Protecting Mobile Agents Using the E-E-D Property............................................................................................. 90

5.5.2. Solution S2: Protecting Mobile Agents Using Hardware-based Secure Computing............................................................... 92

5.5.3. Solution S3: Protecting Mobile Agents Using Obfuscation 93
Table of Contents—Continued

CHAPTER

5.5.4. Solution S4: Protecting Mobile Agents Using Coding Theory and Hardware-based Methods ........................................ 94

5.5.5. Solution S5: Protecting Mobile Agents Using Garbled Circuits and TTPs ............................................................... 95

5.5.6. Solution S6: Protecting Mobile Agents Using Garbled Circuits and Threshold-based Multi-party Computing ....... 97

5.5.7. Solution S7: Protecting Mobile Agents Using Privacy Homomorphisms and Function Composition ..................... 98

5.5.8. Solution S8: Protecting Clueless Mobile Agents Using Environmental Conditions .................................................. 101

5.5.9. Solution S9: Protecting Mobile Agents Using Polymorphism ................................................................................. 103

5.6. Qualitative Evaluation of the Nine Solutions for Mobile Agents Providing Output to Visited Malicious Hosts .............. 104

5.6.1. Requirements for Secure Mobile Agent Solutions .......... 105

5.6.2. Qualitative Evaluation of Nine Solutions ......................... 107

5.6.3. Summary of Qualitative Evaluation of Nine Solutions ..... 110

1.1. Conclusions ............................................................................... 111

6. ABTTP: A TTP-BASED PROTOTYPE IMPLEMENTING THE ACTIVE BUNDLE SCHEME ............................................................. 113

6.1. Introduction ............................................................................... 113

6.2. Threat Model for TTP-based Implementations of the Active Bundle Scheme ................................................................. 114

6.3. ABTTP in the Mobile Agent Architecture Framework ........ 116

x
## Table of Contents—Continued

### 6. Tools and Architectures

6.3.1. Tools and Framework used in the Prototype ...................... 117  
6.3.2. Description of the ABTTP Architecture ............................. 120  
6.3.3. Behavior of the AB Components ........................................ 125  

### 6. Deployment Setups

6.4. Deployment Setups ................................................................. 129  

### 6. Configuration Setup

6.5. Configuration Setup .............................................................. 130  

### 6. Description of the Experiments

6.6. Description of the Experiments .............................................. 133  
6.6.1. Experiment 1: Decryption of an Active Bundle ................. 138  
6.6.2. Experiment 2: Apoptosis of an Active Bundle ................... 139  

### 6. Discussion of the Experiments

6.7. Discussion of the Experiments.............................................. 140  

### 6. Future ABTTP Extensions

6.8. Future ABTTP Extensions ..................................................... 141  

### 6. Conclusions

6.9. Conclusions ........................................................................... 142  

### 7. Privacy and Confidentiality of Autonomous Applications

7.1. Introduction ............................................................................ 144  
7.1.1. Characteristics of Autonomous Applications ................... 146  
7.1.2. Selected Approaches for Protecting Autonomous Applications ........................................... 146  
7.1.3. SAS Computing and SAS Virtual Machine ..................... 148  
7.1.4. Chapter Contributions and Organization ....................... 148  

7.2. Related Work ......................................................................... 150  
7.2.1. Program Obfuscation ....................................................... 150
<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.2. Secure Co-processors</td>
<td>150</td>
</tr>
<tr>
<td>7.2.3. Secure Computing with Encrypted Functions Using Cryptocomputing</td>
<td>151</td>
</tr>
<tr>
<td>7.2.4. Computing of Fixed-functions using Trusted Platform Module</td>
<td>152</td>
</tr>
<tr>
<td>7.3. Proposed Approach to Protecting Confidentiality for Autonomous Applications Using SAS Computing</td>
<td>153</td>
</tr>
<tr>
<td>7.4. Proof of Nonexistence of a Universal Privacy-homomorphic Decryption Function</td>
<td>156</td>
</tr>
<tr>
<td>7.5. Proof of Nonexistence of a Universal Encrypted Decryption Function</td>
<td>163</td>
</tr>
<tr>
<td>7.6. Discussion of the Nonexistence Results</td>
<td>168</td>
</tr>
<tr>
<td>7.7. Conclusions</td>
<td>171</td>
</tr>
<tr>
<td>8. PROTECTING CONFIDENTIALITY OF AUTONOMOUS APPLICATIONS WITH OBFUSCATED CONTROL FLOW GRAPHS</td>
<td>172</td>
</tr>
<tr>
<td>8.1. Introduction</td>
<td>172</td>
</tr>
<tr>
<td>8.2. Control Flow Graphs</td>
<td>174</td>
</tr>
<tr>
<td>8.3. Proposed Solution for Obfuscating CFGs</td>
<td>179</td>
</tr>
<tr>
<td>8.4. Protecting Confidentiality Using the Obfuscated CFGs and Privacy Homomorphisms</td>
<td>182</td>
</tr>
<tr>
<td>8.5. Protecting Confidentiality Using the Obfuscated CFGs and Generic Encrypted Functions</td>
<td>190</td>
</tr>
<tr>
<td>8.6. Discussion of the Results</td>
<td>191</td>
</tr>
</tbody>
</table>
CHAPTER

8.7. Conclusions ................................................................. 192

9. USING ACTIVE BUNDLE SCHEME FOR IDENTITY MANAGEMENT ................................................. 193

9.1. Introduction ........................................................................ 193

9.1.1. Identity Management ...................................................... 193

9.1.2. Identity Management in Cloud Computing ................... 196

9.1.3. Contribution and Chapter Organization ....................... 197

9.2. Related Work ...................................................................... 198

9.3. Proposed Approach for Protecting PII ................................ 198

9.3.1. Characteristics of the Existing Approaches .................. 199

9.3.2. Selected Research Problems ......................................... 199

9.3.3. The Proposed Entity-centric IDM Approach .................. 200

9.3.4. Description of IDM Wallet ........................................... 201

9.3.5. Description of Anonymity-preserving Authentication ..... 202

9.3.6. Simulating the Use of the Proposed Approach for Entity-centric IDM .................................. 206

9.3.7. Characteristics and advantages of the Proposed IDM ..... 207

9.4. Conclusions ...................................................................... 208

10. CONCLUSIONS AND FUTURE WORK ................................. 209

10.1. Conclusions .................................................................... 209

10.2. Directions for Future Work ............................................. 211

10.2.1. Future Work on Improving ABTTP ............................. 211
Table of Contents—Continued

CHAPTER

10.2.2. Future Work on Improving the Active Bundle Scheme ..... 213
10.2.3. Future Work on Improving Security of the Active Bundle Scheme .................................................. 213
BIBLIOGRAPHY ................................................................................................................................. 215
LIST OF TABLES

3.1: Secure computing methods, their support for protecting data and code, and their use of secure computing approaches .......................................................... 33

3.2: Truth table for the circuit (an AND gate) ........................................................ 36

3.3: Garbled truth table for the garbled circuit (a garbled AND gate) ............... 36

3.4: Support for different arithmetic and Boolean operations by selected privacy homomorphisms based on Abelian groups ................................................. 40

5.1: Comparison of Shannon's conventional secrecy model with the mobile agent's secrecy model ................................................................. 68

5.2: Solutions and secure computing methods and approaches used by them...... 89

5.3: Qualitative evaluation of existing solution against requirements ............... 106

6.1: Configuration variables for the ABC component ......................................... 131

6.2: Configuration variables for the AB services component ............................ 132

6.3: Configuration variables for the active bundle client component .................. 133

7.1: Number of hosts required to complete computations for autonomous and non-autonomous applications as the function of the number of layers that implement the application .................................................... 155

7.2: Notation .................................................................................................... 156

8.1: Notation .................................................................................................... 175

8.2: Branching table for Program Π................................................................... 180

8.3: Notation.................................................................................................... 181
LIST OF FIGURES

1.1: A data dissemination scenario ................................................................. 3
1.2: The basic structure of an active bundle ................................................... 9
3.1: Taxonomy of two-party/multiple-interaction secure computing approaches .. 27
3.2: Sequence diagram for secure computing with encrypted data (CED) .......... 27
3.3: Sequence diagram for secure computing with encrypted function (CEF) ..... 28
3.4: Sequence diagram for secure function evaluation (CEDEF) ...................... 31
3.5: A homomorphic cryptosystem .................................................................. 39
3.6: Example of computing with a homomorphic cryptosystem ...................... 40
5.1: Client-server vs. mobile-agent application architecture ............................ 61
5.2: External and internal inputs and outputs of a mobile agent performing computations on a visited host ................................................................. 66
5.3: A taxonomy of secure computing solutions for protecting mobile agents ..... 88
5.4: Sequence diagram for secure computing with encrypted data .................... 95
5.5: Sequence diagram for a two-party/single-interaction algorithm with non-interactive function computing using an encrypted function .................................. 100
5.6: Sequence diagram for encryption of a message using the clueless agent scheme ........................................................................................................... 101
5.7: Sequence diagram for decryption of a message using the clueless agent scheme ........................................................................................................... 102
6.1: Architecture elements of JADE ................................................................. 118
6.2: UML component diagram of the ABTTP prototype: a TTP-based active bundle implementation ................................................................. 121
List of Figures—Continued

6.3: UML activity diagram for constructing an active bundle ...................... 127
6.4: UML activity diagram for enabling an active bundle .......................... 128
6.5: Diagram of the deployment setup of ABTTP ........................................ 130
6.6: Sample screen shots for the executing prototype ................................. 135
6.7: Sample screen shots of an active bundle requesting the user for the destination host .............................................................. 136
6.8: Sample screen shots for the executing prototype for Experiment 1 ...... 137
6.9: Sample screen shots for the executing prototype for Experiment 2 ...... 139
7.1: The superlayer structure of a SAS application .................................... 154
8.1: A simple program $\Pi$ (in C language) that uses branching and loops .... 176
8.2: CFG for Program $\Pi$ ............................................................................ 176
8.3: Flattened program $\Pi$ ........................................................................... 179
9.1: Authentication using third party identity management ........................ 194
9.2: Application-centric vs Entity-centric IDM ........................................... 197
9.3: Structure of IDM Wallet and anonymity-preserving service interaction ...... 201
9.4: Protocol 1(issuing PII by an IdP to an entity) from the Fiat-Shamir identification scheme ................................................................. 203
9.5: Protocol 2 (verifying the identity of an entity) from the Fiat-Shamir identification scheme ................................................................. 204
9.6: Modified Protocol 2 (issuing PII by an IdP to an entity) ....................... 205
CHAPTER 1

INTRODUCTION

Data confidentially and privacy concerns exist whenever sensitive data are created, disseminated, stored, or used. Unauthorized data disclosures are the main data confidentiality threats. The challenge in protecting sensitive data is to satisfy the conflicting needs of data dissemination and data protection, or, in other words, disseminating data while protecting them from unauthorized disclosures [KSW03].

This Thesis proposes the active bundle scheme as a solution for protecting sensitive data from unauthorized disclosure during their dissemination. The scheme protects data throughout their entire lifecycle, from data creation through their dissemination to their evaporation (a partial self-destruction) or apoptosis (the total self destruction).

The scheme is based on a construct named an active bundle, which bundles together sensitive data, metadata, and an embedded virtual machine that makes the bundle “active.”

This chapter includes six sections. Section 1.1 defines the basic terms used in this Thesis, and Section 1.2 provides its motivation. Section 1.3 states the research problem and the goal of the Thesis. Section 1.4 gives an overview of the proposed solution. Section 1.5 discusses contributions of the Thesis. Section 1.6 outlines the organization of the Thesis.
1.1. Definitions of Basic Terms

Privacy is the right of an entity to determine the degree to which it will interact with its environment, including the degree to which the entity is willing to share information about itself with others (cf. [Shir08]). Confidentiality of data is defined as ensuring that data are not made available or disclosed to unauthorized entities, whether human or artificial [Shir08].

As the definitions indicate, privacy is a notion applied to active entities able to interact with their environment, such as individuals or programs, while confidentiality is applied to data [OHRP2009].

Sensitive data (a.k.a. private data) are data that their owners do not want to be made public. (An owner typically is data creator or originator as well.)

Data sensitivity factors (i.e., factors that make data sensitive), include the following (cf. [PfPf07]): (i) being inherently sensitive—e.g., location of defense missiles; (ii) coming from a sensitive source—the source implies that the data are sensitive; (iii) being declared sensitive by the owner; (iv) being in a sensitive context—e.g., an individual’s address in the context of a trial; (v) being a part of sensitive data—e.g., “Chicago” could be a part of a sensitive address data “1904 S. Michigan Avenue, Chicago”; and (vi) being sensitive in relation to previously disclosed data.

Data lifecycle (cf. [BeLi2008]) includes: (i) creating data; (ii) disseminating data from its creator or owner to one or more hosts (including storing data after receiving them, and retrieving data before forwarding them); (iii) disseminating data among hosts; (iv) reduction of data value—through aging or a voluntary evaporation (a partial self-
destruction of data) when threatened with disclosures [LiBh06], etc.; and (v) *apoptosis*—a total self-destruction of data.

*Data dissemination* is the process of transfer (i.e., forwarding) of data from an Entity $G$ (e.g., a human, a host, software, etc.) to a destination Entity $H$ (cf., [BeLi09]). We distinguish two *data dissemination styles* (cf. [PaSS00]): (1) *data push*, when data is sent to a recipient by a particular sender; and (2) *data pull*, when each recipient obtains the data from a data server (an external repository), where data were previously stored by senders.

![Figure 1.1: A data dissemination scenario [BhLi04]](image)

A *guardian* is an entity (either human or not) that accesses data or disseminates it. The owner of sensitive data transfers them to a set of guardians. In turn, a guardian may transfer the received sensitive data to another set of guardians.

Figure 1.1 shows a data dissemination scenario for financial transactions. Guardian 1 is a bank, Guardian 2 is a financial adviser, Guardian 3 is a government agency, Guardian 4 is a mortgage company, and Guardian 5 is a car dealer. The scenario shows dissemination of confidential data from their owner to Guardian 1, who becomes Original Guardian. Original Guardian (a first-level guardian) disseminates sensitive data further to second-level guardians: Guardian 2, and Guardian 3. Guardian 2 disseminates data in turn to third-
level guardians: Guardian 4 and Guardian 5.

Confidentiality-preserving data dissemination assures protection data from unauthorized disclosures during data dissemination. For example, the owner can specify that only his bank (Guardian 1) and his mortgage company (Guardian 4) can access his financial transactions. Confidentiality-preserving data dissemination assures, for example, that Guardian 2 (a second-level guardian) does not access data (although Guardian 2 can transfer them to other guardians), and Guardian 4 (a third-level guardian) can access these data.

One of the basic components used by mechanisms protecting data confidentiality is privacy policies. A privacy policy specifies a set of privacy rules that govern the operations on given data (such as the read, write, or delete operations).\(^1\) An example, a privacy rule is: “The marketing department is allowed to read contact data for the purpose of e-mail marketing.” A privacy rule may define obligations such as: “Delete contact data of the marketing department after 30 days” (cf. [KSW03]).

Privacy policies are verified by evaluating privacy rules, often performed for a certain set of credentials. The evaluation output (resulting from the policy) can be used by an enforcement mechanism deciding whether a requested operation is authorized by the data owner or not.

\(^1\) These policies and rules should be called confidentiality policies and confidentiality rules but historical precedent outweighs naming precision.
A privacy policy may specify purposes of an operation, context constraints, and obligations of entities that use data (cf. [KSW03], [BDMN06]). A privacy policy can specify rules that use histories of data disclosures, e.g., through the use of linear temporal logic [DDLD10].

A widely used method to specify privacy policies is to model policies using policy languages, such as OASIS eXtensible Access Control Markup Language (XACML) [OASI05]. Privacy policy modeling has evolved to include context-aware rules and temporal logic rules. Another method to specify privacy policies is to use Role Based Access Control (RBAC) [KSW03]. RBAC models assign users’ access permissions for resources based on users’ roles. The limitation of this approach is that its privacy policies cannot consider history of accesses to data.

1.2. Motivation for Protecting Confidentiality of Sensitive Data

Protecting confidentiality of sensitive data is a paramount requirement in the cyber world. The great promise of the vision of ubiquitous computing (maybe the most important, comprehensive, and far-reaching challenge in computing today) will not be

---

2 Context constraints are constraints that depend on the environment conditions or factors, such as time, location, program execution state, and access history.

3 Linear temporal logic (LTL) is a system of rules and symbols for representing and reasoning about propositions qualified in terms of time. In LTL, we can encode and formulate events that can happen or can never happen in the future, such as “a condition will eventually be true,” or “a condition is true until another fact becomes true” [Wiki2010a].
realized without confidentiality-preserving data sharing, that is, sharing data while protecting them from unauthorized disclosures.

The ability to secure disseminated data is proving to be a serious challenge. The main challenges are the limitations of current mechanisms for protecting confidentiality of sensitive data during their dissemination. Other challenges include: (1) protecting confidentiality of data used by a set of collaborators who have different levels of security;\(^4\) and (2) assuring authentication and authorization for accessing and modifying data by a set of collaborators.

Confidentiality-preserving information sharing has a broad applicability in numerous areas, including the following:

1) *Pervasive healthcare monitoring.* The use of pervasive medical devices is growing. These devices generate healthcare information about patients that is disseminated and accessed by many healthcare providers. The availability of this information to a healthcare provider could be critical for saving the life of a patient. However, allowing even inadvertent dissemination of this information to a party that may be non-authorized (such as a news company) may be damaging to the patient.

2) *Smart Grid.* The future Smart Grid will allow energy consumers to buy energy from any power company, store energy locally for future use, produce energy, and sell energy back to power companies [Heyd10, VVDR10a, VVDR10b]. In the Smart Grid the distribution of energy is based on the information collected from the different partners (including customers). Unauthorized modification of information exchanged

\(^4\) Collaborators are classified based on *multi-level security (MLS)* [KFE1998].
by the parties (i.e., falsifying the information) or a dissemination of the information to inappropriate parties could disrupt the distribution of energy to customers.

3) **Service Oriented Architecture (SOA):** Current business applications use web services extensively. In SOA [VAMD09], a web application (a service consumer) can use services provided by its partners (service providers). For example, a publisher can use fulfillment web services from Amazon.com. Companies might not use web services if confidentiality of data exchanged between the applications and web services is not protected.

4) **Digital Rights Management:** Digital rights management blocks access to digital content (e.g., documents, movies, etc.) by entities not authorized by the content providers [FFSS01] who sell their digital contents. Unauthorized use of digital content would be a loss for them.

5) **Cloud Computing:** Cloud computing [MKL2009] allows service providers the use of the infrastructure and services offered by clouds. Organizations and companies that deploy their applications on a cloud infrastructure risk having their most valuable business information disclosed by successful attackers who use the same cloud infrastructure.

### 1.3. Problem Statement and Goal

The *research problem* addressed in this Thesis, introduced above, can be summarized as follows:

*Protect sensitive data throughout their entire lifecycle.*
Our research goal is to propose a solution that minimizes risks of unauthorized disclosures of sensitive data throughout their entire lifecycle—including data creation, dissemination, and destruction. Protecting data throughout their entire lifecycle was introduced by us [BeLi09], and later independently proposed under the apt name of *Data Lifecycle Protection* [MacD10].

In this Thesis, we investigate ways of protecting confidentiality of sensitive data. Recall that privacy concerns active entities, while confidentiality concerns data [OHRP2009]. So, indirectly, we investigate protecting privacy of the owner of these sensitive data.

Our focus is on protecting confidentiality of sensitive data throughout their lifecycle. We do not consider authenticity or legitimacy of sensitive data, and do not deal with piracy against sensitive data. Possible solutions for data authentication could use, e.g., digital certificates for data. Solutions against piracy could be based, e.g., on watermarking and forensic approaches.

We abstain from the discussion of security issues other than confidentiality (in the classic definition, *security* consists of confidentiality, integrity, and availability [PfPf07]).

### 1.4. The Proposed Approach and Its Overview

In current confidentiality (and security) solutions, data are commonly considered to be passive entities that cannot protect themselves. As a consequence, they require the use of another active and trusted entity to protect them (e.g., a trusted processor, a trusted memory module, or a trusted third party). We propose an approach that transforms passive data into an active entity by encapsulating them within *active bundles* (analogous to...
making data in programming languages “active” by encapsulating them in abstract data types). To be precise, data becomes active (hence, able to self-protect) by becoming an inextricable part of an active bundle.

The active bundle scheme [BeLi09] is based on a software construct named an active bundle, which—as illustrated in Fig. 1.2—bundles together sensitive data, metadata, and a virtual machine specific to the bundle. Metadata contain information about data carried by the active bundle. They specify policies for dissemination of data by active bundles (the policies include access control rules for these data). The virtual machine (VM) controls its active bundle and enforces the policies specified by metadata. Sensitive data, metadata, and the VM of an active bundle are collectively called a payload of the bundle.

The creators of an active bundle are the entities (either human or artificial) that created its payload. The owners of an active bundle are the entities (either human or artificial) that hold usage rights for its payload. These rights can be either formal (e.g., a patent) or informal (e.g., a trade secret). It is possible that an owner owns only a part of a bundle’s payload. For example, a human creator might own only data (a part of a bundle’s payload) but its metadata or the VM might belong to, e.g., the bundle dissemination middleware. As another example, a creator might own no portion of the payload. (This is analogous to
a very common situation when patent rights belong to a company, not to its creator, the
inventor, who is a company’s employee.)

The active bundle scheme protects confidentiality of sensitive data by: (i) enforcing
privacy policies specified within a bundle’s metadata, (ii) activating protection
mechanisms (embedded within a bundle’s VM) when data are tampered with, and (iii)
recording for future audits all activities involving a bundle (e.g., a transfer of the bundle
from a source to a destination).

The active bundle scheme protects sensitive data as long as they are accessed through
an active bundle. Sensitive data or their parts are not protected once they are given as an
output to any untrusted entity (since they leave the sphere of control of the scheme).

As will be described, an active bundle offers mechanisms (such as implementation of
disclosure rules, evaporation, and apoptosis) to protect its payload. It requires, in turn,
certain mechanisms to protect its own confidentiality, i.e., protecting confidentiality of its
VM and the VM’s execution from untrusted hosts. For instance, an untrusted host viewing
an unprotected execution of a bundle’s VM can extract sensitive data that it is not
authorized to access.

Protecting the VM and its execution is the most challenging element of protecting an
active bundle. The reason is that a VM’s execution requires hardware that cannot be a part
of an active bundle.

We identified the following three strategies for protecting confidentiality of complete
active bundles (including their VMs with the VMs’ executions):
1) The ideal strategy: It is a mental experiment, an unattainable benchmark that can provide useful practical hints. If we had teleportation (so popular in science fiction), an active bundle could include not only data/software but an embedded hardware as well. The ideal software/hardware bundle would be teleported to a destination host, and the bundle’s VM would execute on the bundle’s own embedded and trusted hardware.

2) The best available practical strategies: A bundle contains only software (including the bundle’s VM) that can be disseminated among hosts. The best practical strategy available to us currently to assure bundle confidentiality requires using Trusted Third Parties (TTPs) for the VM execution.

3) The dominant practical strategies: It will improve upon the best available practical strategy by, among others, being independent of TTPs, and attempting to approach the performance that the ideal strategy would provide.

The best (currently) available practical strategy developed and implemented by us is presented in the Thesis (in Chapter 6). We also investigate (in Chapters 7 and 8) two different candidate approaches for dominant practical strategies, and provide groundwork for them. However, their full development extends well beyond the scope of this Thesis.

1.5. Main Thesis Contributions

Our thesis is:

Data can protect themselves from unauthorized accesses by malicious hosts.

The main contributions of this Thesis include the following:
1) We propose the **active bundle scheme** for protecting sensitive data throughout their lifecycle. As was indicated above, the scheme protects sensitive data from unauthorized disclosures.

2) Mobile agents [Lan98] are identified by us as a candidate foundation for protecting confidentiality of data and code, and thus can be used to implement active bundles.

   We answer the following two questions: (i) Are there any available solutions for protecting confidentiality of code and carried data of mobile agents providing output to visited hosts that satisfy the two critical confidentiality-related mobile agent properties: computation autonomy and data autonomy? (ii) Are any of these solutions practical?

   The answer to the first question is “Yes,” but the response to the second question is “No.”

3) We developed a TTP-based prototype of the active bundle scheme. The prototype demonstrates practicality of the scheme by being able to efficiently protect confidentiality of sensitive data when unauthorized entities attempt to access these data.

4) We answer the following two questions:

   (i) Is there a universal privacy-homomorphic\(^5\) decryption function?

   (ii) Is there a secure autonomous sequential virtual machine for an encrypted decryption function?

---

\(^5\) A *privacy homomorphism* (i.e., a *homomorphic cryptosystem*) is an encryption transformation that allows computations with encrypted data; a *privacy-homomorphic function* is a function computed using a privacy homomorphism.
5) We pioneer—to the best of our knowledge—use of secure computing for program obfuscation.

6) We present a sample application of active bundles for identity management.

1.6. Thesis Organization

The thesis is organized as follows. Chapter 2 discusses related work on confidentiality of disseminated data. Chapter 3 introduces secure computing, which is used by us as a component of solutions presented in Chapters 5 and 7. Chapter 4 describes the active bundle scheme. Recall that our contributions include answering two questions (listed in Item 2 in Section 1.5) related to availability and practicality of specific confidentiality-preserving solutions for mobile agents. Chapter 5 describes our literature search and the answers obtained in the process. Chapter 6 presents the TTP-based prototype implementing the active bundle scheme. Chapter 7 discusses protecting confidentiality of autonomous applications using secure computing. Chapter 8 introduces our preliminary results on protecting confidentiality of autonomous applications with obfuscated control flow graphs. Chapter 9 discusses a sample application of the active bundle scheme for identity management. Chapter 10 concludes the Thesis, and indicates directions for extending this research.
CHAPTER 2

LITERATURE REVIEW ON PROTECTING CONFIDENTIALITY
OF DISSEMINATED SENSITIVE DATA

2.1. Introduction

Researchers have proposed different mechanisms to protect confidentiality of sensitive
data, including cryptography, steganography, obfuscation, and bundle dissemination. In
this chapter we present a review of the literature on solutions for protecting confidentiality
of sensitive data disseminated to a set of entities.

We classify the solutions into two classes. The classification is based on whether the
set of entities receiving disseminated data is enumerable or not enumerable. More
precisely, the classes are: (i) a data owner can enumerate all entities that should be able to
access the disseminated data; (ii) a data owner cannot enumerate all entities that should be
able to access the disseminated data.

We discuss below a set of solutions for each of the two classes of solutions. Section 2.2
discusses solutions for enumerable receiving entities, and Section 2.3 discusses solutions
for non enumerable receiving entities. Section 2.4 concludes the chapter.

2.2. Confidentiality Protection Solutions for Enumerable Receiving Entities

Let us consider the class of solutions for protecting confidentiality of sensitive data
where entities receiving data are known a priori. The owner can use an encryption
mechanism to encrypt data with keys in such a way that each entity can decrypt data that it is allowed to access by confidentiality restrictions (e.g., policies). In this section, we describe the following three solutions for protecting confidentiality of sensitive data: using multiple keys, using policies and TTPs, and using trusted VMs.

2.2.1. Protecting Confidentiality of Sensitive Data Using Multiple Keys

Digibox is a cryptographically protected container with data and controls that enforce rights to these data. The DigiBox scheme [Sibe95] organizes data into items. Each data item is assigned an encryption key, and is encrypted with this key. Each entity receiving a DigiBox can decrypt only the data item that it is allowed to access because it has the decryption keys for decrypting only these items.

The DigiBox scheme focuses only on secure transmission of sensitive data between hosts. Its weakness is that a destination host is not restricted in any way from disseminating a received DigiBox further without approval (or even knowledge) of its owner. In contrast, our active bundle scheme prevents such unauthorized activities.

2.2.2. Protecting Confidentiality of Sensitive Data Using Policies and TTPs

Park, Sandhu, and Schifalacqua [PaSS00] present a taxonomy of architectures for managing access control and dissemination control for sensitive data and digital content. The architecture classification is based on using or not using metadata for access and
dissemination control, using or not using VMs located on the hosts, and the dissemination style.\(^6\)

The authors note that using a host’s VM reduces the security of digital content. Solutions using a VM at the receiving host assume that VMs at all hosts are fully trusted. This assumption is too strong to be generally acceptable. In contrast, our active bundle scheme requires only that an active bundle itself, not each host, includes a trusted VM.

### 2.2.3. Protecting Confidentiality of Sensitive Data Using Trusted VMs

Borders et al. [BVBP09] propose Storage Capsules for protecting confidentiality of files on a personal computer. Storage Capsules are encrypted file containers that can be used on untrusted computers (even the ones infected by malware) to securely view and edit sensitive files without malware being able to steal confidential data. The approach is based on using a trusted VM to access securely a Storage Capsule. The solution takes a checkpoint of the current system state and disables output devices before switching to a trusted VM and allowing access to a Storage Capsule. Writes to the Storage Capsule are then sent to the trusted VM. When the user finishes editing the Storage Capsule, the system is restored to its original state, and device output resumes normally. The trusted VM declassifies the Storage Capsule by re-encrypting its contents, and exports it for storage in a low-integrity environment.

\(^6\) The dissemination styles are [PaSS00]: (i) message push, in which a user receives a copy of the digital container; and (ii) external repository (which we call message pull), in which a user gets a digital container from a repository.
The Storage Capsule and the active bundle schemes have a common objective: to protect confidentiality of sensitive data. The two schemes differ in the following: (i) the Storage Capsule scheme is based on using a trusted VM that has exclusive access to Storage Capsules, while the active bundle scheme requires using a VM embedded within the active bundle; (ii) the Storage Capsule scheme uses encryption mechanisms to protect sensitive data, while the active bundle scheme uses control access policies and protection mechanisms (self-integrity check, evaporation, and apoptosis) to manage access to sensitive data; (iii) an entity receiving a Storage Capsule needs to have an appropriate setup that can open the Storage Capsule (the setup must include a trusted VM and a VM monitor), while the active bundle scheme does not require such a setup (it only requires the host to execute the VM included in the bundle).

2.3. Confidentiality Protection Solutions for Non-Enumerable Receiving Entities

Let us consider now the class of solutions for protecting confidentiality of sensitive data where entities receiving data are not known a priori. In this section, we describe the following five solutions for protecting confidentiality of sensitive data: using sticky policies and identity-based encryption, the Vanish system, data tethers, using activation time, and using bundles and active bundles.
2.3.1. Protecting Confidentiality of Sensitive Data Using Sticky Policies and Identity-based Encryption

TPM is a security hardware component. It can facilitate numerous important capabilities, such as authentication, hardware-based encryption, digital signing, secure key storage, and attestation of software installed on hardware. For encryption and signing, TPM uses keys stored in a protected hardware storage [TCG10].

TPM enables building trustworthy computing relationships. For instance, it can be used to prevent tampering software from exploiting the chain of trust [Parn10]. For building a chain of trust, TPM evaluates a hash function for the host’s BIOS, and compares this hash value to a value stored by the TPM. If both values are equal, then the BIOS can be trusted (else, it can not). If the BIOS is trustworthy, it can in turn be used to determine the trustworthiness of the operating system (OS) loader. If the OS loader is trustworthy, it can in turn be used to determine the trustworthiness of the OS [Parn10].

Casassa Mont et al. [CaDA03] propose a technique for enforcing confidentiality protection during dissemination of sensitive data (such as digital identities) using sticky policies. *Sticky policies* are policies that are inseparable from the associated data. Casassa Mont et al. investigate how to implement sticky policies, that is, how to make data and their policies inseparable in a way that an attacker cannot access data without satisfying their privacy policies. They identify two approaches for associating data with their related policies: (i) use of identifier-based encryption, and (ii) use of a Trusted Platform Module (TPM).

In the first approach, data is sent from their owner to a receiver who may disseminate them further. The owner constructs encryption keys using a receiver’s identifier, constraints and conditions defined by the data access policies. The owner also encrypts
data, and sends data and their associated data access policies to a receiver. The receiver provides its credentials and data access policies to a trusted authority (TA). The TA computes the decryption key using the receiver’s credentials and the sticky policies. Then, the TA sends the key to the receiver. The decryption key can decrypt data only if the receiver provided the TA with the appropriate credentials for the sticky policies associated with these data.

The second approach—using TPM—assumes that the operating system is trusted by TPMs, and—furthermore—enforces the data access policies associated with a given data set. This approach can be attacked in at least these three ways:

1) An authorized receiver is not restricted in any way from further dissemination of received data without approval (or even knowledge) of their owner. In contrast, our active bundle scheme prevents such unauthorized activities.

2) If data and policies are received by a host that has no TPM and no policy engine, data cannot be accessed. In contrast, in our scheme we do not require hosts to have their own policy enforcement engines; instead, it is as a part of the bundle’s VM.

3) A policy enforcement engine available at a host could be changed by the host in such a way that the host can access data against the “will” of the data access policies. In contrast, in our scheme hosts do not own policy enforcement engines; instead, the privacy policy engine is a part of the bundle’s VM (our privacy policies include data access policies).

A different solution for a policy-based cryptographic scheme was developed by Bagga and Molva [BaMo05] which uses identity-based encryption. The scheme allows: (i) performing data encryption operations using privacy policies formalized as Boolean
expressions, and (ii) performing data decryption using policies and digital credentials of an entity that wants to decrypt these data. The approach uses identity-based encryption scheme.

### 2.3.2. Protecting Confidentiality of Sensitive Data Using the Vanish System

Geambasu et al. [GKLL09] present the Vanish system which increases the degree of data confidentiality by using self-destructing data. (We used data self-destruction ideas for active bundles, originally presented by Lilien and Bhargava [LiBh06], at least a year earlier [LiBh08].) Vanish uses the following steps to encapsulate data $d$. It picks a random number $K$ as a key, encrypts $d$ using $K$ to get $C=E_K(d)$. Then, it splits $K$ into $N$ shares $K_1, \ldots, K_N$ using the threshold secret sharing technique [Sham79]; the technique uses a threshold $t$ that defines the minimum number of (key) shares required to reconstruct $K$. The reconstructed key $K$ is then used to decrypt $d$: $d=D_K(C)$.

Vanish provides consumer and producer plug-ins for the Firefox web browser. A consumer plug-in is required by Vanish (which is its proxy). A user enters data on a web page form. Data are encrypted to create a Vanishing Data Object (VDO). During this data encryption, Vanish splits the decryption key $K$ into shares. The decryption key shares are stored in a distributed hash table (DHT) system (such as Vuze [FPJK07]), where each share $K_i$ is stored at a location $l_i$. A property of DHT is that data are stored (using the tuple <index, value>) redundantly on a set of nodes (Vuze uses a 20-fold replication for each data item). The redundant VDO key shares stored in a DHT expire and are removed after a timeout $W$. This means that data in Vanish become useless after a timeout since they can no longer be accessed after their key shares disappear.
The limitations of the Vanish solution include the following:

1) *Data may be republished in an unauthorized way*: An owner of a host uses a consumer plug-in required by Vanish (the plug-in is a proxy for Vanish). The host’s owner can modify the code of the plug-in in a way that allows him to learn the secret encryption/decryption key $K$ once it is fetched by the plug-in and used to access data (e.g., the host’s owner can modify the plug-in by adding a statement to the decryption function included in the plug-in to print the decryption key). Then, the host’s owner can republish data. Republished data do not vanish (as desired) when their keys expire.

In contrast to Vanish, the active bundle scheme does not use components pre-installed on hosts receiving data; instead, our scheme uses an embedded VM.

2) *Limited capability for data self-destruction*: Data self-destruction in Vanish is only timeout-driven. Before key share timeout occurs, data undergoing attacks (by entities that got hold of their enclosing VDO) are protected only by encryption. In contrast to Vanish, our active bundle scheme allows data self-destruction when its bundle is attacked (there is no dependency on any timeout).

The authors evaluated security and performance of Vanish by experimenting with a deployment of its implementation to 200 hosts in the Amazon cloud computing system. The experiments included: (i) varying the number of key shares $N$, (ii) varying the threshold value $t$, and (iii) checking effects of availability of hosts storing shares of decryption keys.

The evaluation shows that the Vanish scheme is practical and resilient to several types of attacks, including the store sniffing attacks (an attacker compromises some of the nodes
used to store key shares), and Sybil attacks (in which relatively few malicious nodes attack a system by assuming a large number of identities). The probability that such attackers illegitimately access a VDO is estimated.

2.3.3. Protecting Confidentiality of Sensitive Data Using Data Tethers

Beaumont-Gay et al. [BeER07] propose using Data Tethers—a solution that uses policies based on environmental factors to determine when data can be used and when they can be removed from a host entirely. Data Tethers use a security server to store decryption keys for encrypted data. They also employ policy engines that evaluate policies using environmental input. An environmental input could be, for example, the location from which data are accessed, or a certain clock time. Data Tethers also use a process management tool to track usage of disclosed sensitive data by processes, and to kill all processes that use or used these data.

The disadvantage of this solution is that it relies on a TTP: a security server storing decryption keys. The simplest implementation of our active bundle scheme also utilizes a TTP; however, we are working on future and more advanced implementations that will not require any TTPs.

2.3.4. Protecting Confidentiality of Sensitive Data Using Activation Time

Casassa Mont et al. [CaHS02] propose a solution for document dissemination. A document owner plans to send it to a set of recipients. He generates a symmetric key $SK$. Then, he encrypts the document using $SK$. Next, he encrypts $SK$ using an encryption
key based on the planned date and time of the disclosure of the document (e.g., “GMT201009101200” is an encryption key for documents to be disclosed at 12:00 a.m. on September 10, 2010). Next, the encrypted document is distributed to intended receivers.

The solution uses a time vault service, which continually generates and publishes decryption keys associated with the current date and time (obtained by a trusted clock).

A receiver can open a document only with a decryption key, provided to the document receiver not with the document but later, at the document disclosure time. The receiver gets this key from the time vault service. Then, the receiver uses the decryption key (obtained from the time vault) to decrypt the symmetric key. Next, the receiver decrypts the document using the symmetric key.

In contrast, in the active bundle scheme the rules to disclose data are not limited to using timing conditions only. Instead, data from an active bundle are disclosed to any receiver that satisfies the privacy policies included in the bundle.

2.3.5. Protecting Confidentiality of Sensitive Data Using Bundles and Active Bundles

Lilien and Bhargava [LiBh06] present an approach for protecting disseminated sensitive data using the bundle scheme. A bundle packages sensitive data and metadata associated with these sensitive data. Metadata includes polices, including policies for data access and dissemination. The scheme assumes that hosts receiving bundles can be trusted to enforce privacy policies defined in the bundle’s metadata.

Active bundles, proposed in this Thesis (and earlier in Reference [BeLi08]), remove the trusted host assumption, and associate a VM with the bundle, making it “active.” The VM
enforces privacy policies, and controls several protection mechanisms. Chapter 4 provides details of the active bundle scheme.

2.4. Conclusions

Various solutions for protecting confidentiality of sensitive data are proposed in the literature. This chapter classifies the available solutions into two categories: the class of confidentiality-preserving data dissemination solutions for enumerable receiving entities, and the class of confidentiality-preserving data dissemination solutions for non-enumerable receiving entities. The first class includes the following confidentiality-protecting solutions for sensitive data: protection using multiple keys, protection using policies and TTPs, and protection using trusted VMs. The second class includes the following confidentiality-protecting solutions for sensitive data: using sticky policies and identity-based encryption, the Vanish system, data tethers, using activation time, and using bundles and active bundles.
3.1. Introduction

Secure computing\textsuperscript{7} is computing which prevents any unauthorized disclosure of data or functions (cf. [Yao82]). (For historical reasons, to be consistent with the prevailing practice, we use the term secure computing while in fact it is confidentiality-preserving computing.)

We can consider three secure computing approaches: secure computing with encrypted data (CED), secure computing with encrypted function (CEF), and secure computing with encrypted data and encrypted function (CEDEF).\textsuperscript{8} In general, these approaches are used for solving a number of secure computing problems (including confidentiality-preserving\textsuperscript{9} data storage outsourcing and confidentiality-preserving distributed data mining [Kole09]).

This chapter describes the three secure computing approaches (CED, CEF, and CEDEF) and ten methods used for protecting data and function confidentiality. It is organized as follows. Section 3.2 discusses approaches for secure computing. Section 3.3

\textsuperscript{7} In the literature both terms secure computing and secure computation are used. We chose the former to emphasize the action of computing.

\textsuperscript{8} In the literature, “SFE” is widely used (e.g., [McDo2006]). We changed the acronym to “CEDEF” for consistency with the associated acronyms “CED” and “CEF.”

\textsuperscript{9} In the literature the term privacy-preserving is commonly used to refer to preserving confidentiality of data. In this thesis the terms privacy-preserving and confidentiality-preserving are considered equivalent.
discusses methods that implement the three named secure computing approaches. Section 3.4 concludes the chapter.

3.2. Three Secure Computing Approaches

We divide algorithms for secure computing along two dimensions: (i) two-party or multi-party; and (ii) single-interaction or multiple-interaction. For us, two-party and multi-party are a dichotomic classification. Two-party algorithms limit the number of interacting parties to two. The multi-party algorithms involve more than two parties.

In single-interaction algorithms, one interaction between partners (i.e., a unidirectional transfer of data from one partner to another) is sufficient to perform computing, while multiple-interaction algorithms use many interactions in their computing.

Fig. 3.1 shows taxonomy of two-party/multiple-interaction secure computing approaches for protecting confidentiality of data that are discussed next. (Only the two-party/multiple-interaction secure computing approaches are discussed here to simplify the introduction to secure computing problems. We could show similar two-party/single-interaction approaches as well as multi-party/multiple-interaction approaches.\footnote{Note that multi-party/single-interaction approaches do not exist. A single interaction is among just two parties, so they are two-party/single-interaction approaches.})

Various secure computing approaches are discussed in more detail later (e.g., in Subsection 3.3.4 we describe obfuscation which could be implemented as two-party/single-interaction secure computing).

\footnote{To be precise, “multiple” means here “two.”}

\footnote{Note that multi-party/single-interaction approaches do not exist. A single interaction is among just two parties, so they are two-party/single-interaction approaches.}
We discuss now the three secure computing approaches for protecting confidentiality of data.

### 3.2.1 Protecting Confidentiality of Data Using Secure Computing with Encrypted Data (CED)

The problem of secure computing with encrypted data \((CED)\) can be stated as follows [McDo06]: Alice has a function \(f\), and Bob has an input \(x\). Bob wants to know the result of \(f(x)\). Alice does not want Bob to know anything about \(f\), and Bob does not want Alice to know anything about \(x\) and \(f(x)\).

To solve the CED problem, we propose the approach realized by the following simple generic protocol, shown in Fig. 3.2: Bob encrypts \(x\) (using the encryption function \(E\)), and

---

\(^{12}\) “CED” rather than “SCED” is used in the literature (e.g., [McDo2006]).
sends $E(x)$ to Alice; Alice computes $f(E(x))$, and sends $f(E(x))$ to Bob; Bob decrypts $f(E(x))$ and gets $f(x)$.

The following property:

$$D( f(E(x)) ) = f( D(E(x)) ) = f( x)$$

must underlie this particular protocol, allowing Bob to decrypt $f(E(x))$ to get $f(x)$.

For a practical solution, this property must be implemented in some way. Some of the potential ways may be based on using privacy homomorphisms, coding theory, or other methods (cf. Section 3.3). As an example, polynomial functions (e.g., $f(x) = 4x^3 + 5x^2 + 6$) could be implemented using a homomorphic cryptosystem that supports both addition and multiplication.

3.2.2 Protecting Confidentiality of Functions Using Secure Computing with Encrypted Functions (CEF)

The problem of secure computing with encrypted function (CEF\textsuperscript{13}) can be posed as follows [McDo06]: Bob has a function $f$, and Alice has an input $x$. Bob wants to know the value of $f(x)$. Bob does not want Alice to know $f$, and Alice does not want Bob to know $x$.

![Figure 3.3. Sequence diagram for secure computing with encrypted function (CEF)](image)

\textsuperscript{13}“CEF” rather than “SCEF” is used in the literature (e.g., [McDo2006]).
To solve the CEF problem, we propose the approach realized by the following simple generic protocol, shown in Fig. 3.3: Bob encrypts his function $f$ to get $E(f)$, and sends $E(f)$ to Alice; Alice computes $E(f)(x) = E(f(x))$, and sends $E(f(x))$ to Bob; Bob decrypts $E(f(x))$ and gets $f(x)$.

The following property:

$$D(E(f)(x)) = f(x)$$

must underlie this protocol, allowing Alice to calculate $E(f)(x)$ when she knows only $E(f)$ but not $E$ or $f$.

For a practical solution, this property must be implemented in some way. Some of the potential ways may be based on using garbled circuits (cf. Section 3.3.2). As an example, binary functions could be implemented using garbled circuits (e.g., $f(x) = x \ AND \ 01$, where $\ AND$ is a bitwise operation, and $01$ is a Boolean constant).

Since CEF was implemented for the first time using garbled circuits, CEF (or “secure computing with encrypted function”) is commonly—though not quite correctly—used in the literature to refer to just the garbled circuit implementations.

CEF works if $f(x)$ does not have an inverse function.\footnote{Let $f$ and $g$ be two functions. If $f(g(x)) = x$ then $f$ is the inverse of $g$ (cf. [BrHe2010]). Otherwise, Bob could infer Alice’s input $x$. Consider Fig. 3.3. Bob knows $f$, $E(f)$, and $f(x) = D(E(f)(x))$. Since Bob knows $f$, he can calculate $f(x')$ for various values of $x'$. Eventually, Bob will find $x'$ such that $f(x') = f(x)$. If $f$ has an inverse function, then Bob knows that $x = x'$ (there is no $x' \neq x$ such that $f(x') = f(x)$).

In contrast, if $f$ does not have an inverse function, Bob cannot be sure that $x' = x$. For example, let $f(x) = x^2$. Then, for $x \neq 0$, $f(x) = x^2$ has two values that produce the same value of $f(x) = x^2$. For instance, if Bob receives the function value $f(x) = 4$ calculated by Alice,
he will not know if \( x \) was equal to \( x' = 2 \) or to \( x'' = -2 \). In such a case, Alice’s input value \( x \) is protected from Bob, as claimed above.

### 3.2.3 Protecting Confidentiality of Data and Functions Using Secure Computing with Encrypted Data and Encrypted Functions (CEDEF)

The problem of secure computing with encrypted data and encrypted functions (CEDEF\(^{15}\)) can be posed as follows [McDo06]: Alice has an input \( x \), and Bob has an input \( y \), and a function \( f \). Both Alice and Bob want to know the result of function \( f \) applied to both inputs, i.e., \( f(x, y) \). Bob does not want Alice to know \( y \) or \( f \), and Alice does not want Bob to know \( x \).

There are two variants of CEDEF. They use either a public function [GMW87] or a private function [SaYo99], where the former is a function known to both parties (Bob and Alice), and the latter is known to only one party (Bob).

Only one party, either only Alice or only Bob, obtains \( f(x, y) \) (it is the party that holds the encryption key \( E \) used to encrypt the data of both parties and function \( f \)). The party obtaining the result may share it with the other party (cf. [SaYo99, McDo06]).

\(^{15}\) To be consistent with earlier acronyms (“CEF” and “CED”) we use “CEDEF.” In the literature (e.g., [McDo2006]), “secure function evaluation” is commonly used to refer to secure computing with encrypted data and encrypted function.
To solve the CEDEF problem, we propose the approach realized by the following simple generic protocol, shown in Fig. 3.4: Bob encrypts \( y \) and \( f \) (using the encryption function \( E \)), and sends \( E(y) \) and \( E(f) \) to Alice; Alice computes \( E(f)(E(x), E(y)) \); and sends \( E(f)(E(x), E(y)) \) to Bob; Bob decrypts \( E(f)(E(x), E(y)) \) to get \( f(x, y) \).

One of the following properties:

(i) \( D( E(f)(E(x), E(y)) ) = f(x, y) \)

or

(ii) \( D( E(f)(E(x), E(y)) ) = f(x, y) \)

must underlie this particular protocol, allowing Alice to calculate \( f(x, y) \) even though she knows only \( E(y) \) but not \( y \). The protocol that we described above relies on Property (i) but a similar generic protocol that instead relies on Property (ii) could be used.

For a practical solution, one of these properties must be implemented in some way. Potential ways may be based on using coding theory, privacy homomorphisms, or other methods (all discussed in Section 3.3).

CEDEF can be solved with simpler single-round protocols (i.e., two-interaction protocols) or more complex multiple-round protocols (where a round consists of two
interactions: the first interaction is a request message, and the second interaction is the reply message).

We choose above a simpler single-round generic protocol. An example of implementation of a single-round protocol for secure evaluation of circuits is provided by Sander and Young [SaYo99]. The protocol uses Property (i). (Most of the more complex multiple-round generic protocols for solving CEDEF (e.g., [GMW87, BMR90] use Property (ii).)

CEDEF works if $f(x, y)$ does not have an inverse function. Otherwise Bob could infer Alice’s input $x$. Consider Fig. 3.4. Bob knows $y$, $f$, $E(y)$, $E(f)$, and $f(x, y) = D(E(f)(E(x), E(y)))$. Since Bob knows $f$ and $y$, he can use it to calculate $f(x’, y)$ for various values of $x’$. Eventually, Bob will find $x'$ such that $f(x’, y) = f(x, y)$. If $f$ has an inverse function, then Bob knows that $x = x’$ (there is no $x’ \neq x$ such that $f(x’, y) = f(x, y)$). In contrast, if $f$ does not have an inverse function, Bob cannot be sure that $x' = x$.\footnote{In this case, the attacker (“Bob”) cannot distinguish between the correct and wrong values. He knows only the set of values including the correct one. Security in this case, based on distinguishability, is weaker than \textit{semantic security}, where an attacker has a very small probability of finding the correct value.}

\section{Methods for Secure Computing}

This section describes methods used for protecting confidentiality of computing with data and functions. Most of the methods are secure computing methods, that is, they implement one or more of the secure computing approaches (CED/CEF/CEDEF) discussed in the previous section.
Methods presented in this section can be used for solving the multiple-interaction or single-interaction secure computing problems. Table 3.1 shows for each method whether it protects data, functions, or both. The entry “depends” means: “depends on the particular mix of component methods used.”

Table 3.1 also shows which of the secure computing approaches, if any, is/are used in the given method. Note that some discussed methods—e.g., the trust-based methods—provide protection for data and/or function but are not based on any of the three secure computing approaches.

<table>
<thead>
<tr>
<th>Secure Computing Method</th>
<th>Protects Data</th>
<th>Protects Code (Functions)</th>
<th>Secure Computing Approach Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding theory</td>
<td>no</td>
<td>yes</td>
<td>CEF</td>
</tr>
<tr>
<td>Garbled circuits</td>
<td>yes</td>
<td>yes</td>
<td>CEF, CEDEF</td>
</tr>
<tr>
<td>Hardware-based methods</td>
<td>yes</td>
<td>yes</td>
<td>CEDEF</td>
</tr>
<tr>
<td>Obfuscation</td>
<td>yes</td>
<td>yes</td>
<td>Note 1</td>
</tr>
<tr>
<td>Privacy homomorphisms</td>
<td>yes</td>
<td>no</td>
<td>CED</td>
</tr>
<tr>
<td>TTP-based methods</td>
<td>yes</td>
<td>yes</td>
<td>Note 1</td>
</tr>
<tr>
<td>Threshold-based methods</td>
<td>yes</td>
<td>no</td>
<td>Note 1</td>
</tr>
<tr>
<td>Function composition</td>
<td>no</td>
<td>yes</td>
<td>CEF</td>
</tr>
<tr>
<td>Hybrid methods</td>
<td>depends</td>
<td>depends</td>
<td>Note 2</td>
</tr>
<tr>
<td>Other methods</td>
<td>depends</td>
<td>depends</td>
<td>Note 2</td>
</tr>
</tbody>
</table>

Notes:
1. Does not use any secure computing approach (neither CED, CEF nor CEDEF).
2. Use of zero or more secure computing approaches depends on the particular combination of component methods used by a particular hybrid/other method.
3.3.1. Coding Theory

Coding theory is the study of methods to transfer data efficiently and accurately from a source location to a destination location [HHLL91]. Typically, a data transfer medium is called a channel (e.g., a landline or a wireless link). To achieve efficiency, a code word should be represented by the minimum number of bits. To achieve accuracy, error control techniques are used for detecting and correcting errors that occur when data are transferred or stored. These techniques add controlled redundancy to messages to improve the chances of recovering the original sent message from a distorted received message [Lour01].

As an example, coding theory is used for protecting data exchanged by communicating parties in McEliese public cryptosystem [97]. Coding theory first transforms $M$ to $M'$, where $M' = M \cdot P$ ($P$ is a random matrix). Then, $M'$ is sent to a destination host. Next, the receiving host computes $M'$ using its input to produce output $y'$, and sends $y'$ to the originator. The originator uses the inverse of $P$, $P^{-1}$, to compute $y$ from $y'$.

3.3.2. Garbled Circuits

Yao [Yao86] proposes a general solution for a two-party secure function evaluation when the attacker is inquisitive.\textsuperscript{17} The basic idea is to provide a method to compute

\textsuperscript{17} An inquisitive adversary (originally named honest-but-curious [LiLi05]) follows the protocol, but attempts to learn more information than allowed.
a garbled (or scrambled) circuit evaluating a function in such a way that the input values of two parties are never revealed to the other party.

In more detail, the proposed solution is as follows. Let $f$ be a polynomial-time function computable by circuit $C$, and let $x$ and $y$ be the inputs of the two parties. Circuit $C$ is computed gate by gate, using the inputs provided by the parties, until the output of the final gate is generated. The solution relies on having the circuit encrypted, and on an interactive protocol that is used by the parties to decrypt the output to the appropriate party. Yao [Yao86] did not show how to implement his protocol, but other researchers did so. In the following, we use the technique provided by Lindell and Pinkas [Lin09] and the protocol described by Goldreich [Gol04].

Implementation of a garbled circuit for $n$ parties that can be used for protecting circuits involves the following steps (illustrated for $n = 2$ parties):

1) *Constructing a garbled circuit:* The originator (the owner of the circuit) constructs the garbled circuit as follows. Let $g$ be a gate, $w_1$ and $w_2$ its input wires, and $w_3$ its output. Each wire represents a bit that takes values 0 or 1. Let $w'_i$ and $w'_j$ be two keys (which are random values) associated with two inputs, where $i$ is the wire label, and $\sigma$ and $\tau$ denote one of the two binary input values (0 or 1). The method involves providing a garbled computation table (a garbled truth table) that maps the keys for inputs into an encryption of the output. Table 3.2 shows a truth table for a circuit; in this case, the circuit is an AND gate.
Table 3.2. Truth table for the circuit (an AND gate)

<table>
<thead>
<tr>
<th>(w_1) IN</th>
<th>(w_2) IN</th>
<th>(w_3) OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3. Garbled truth table for the garbled circuit (a garbled AND gate) [Lin09]

<table>
<thead>
<tr>
<th>(w_1) IN</th>
<th>(w_2) IN</th>
<th>(w_3) OUT</th>
<th>Garbled Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(w_0^1)</td>
<td>(w_0^2)</td>
<td>(w_0^3)</td>
<td>(E_{w_0^1}(E_{w_0^2}(w_0^3)))</td>
</tr>
<tr>
<td>(w_1^0)</td>
<td>(w_1^1)</td>
<td>(w_1^3)</td>
<td>(E_{w_1^0}(E_{w_1^1}(w_1^3)))</td>
</tr>
<tr>
<td>(w_1^0)</td>
<td>(w_1^2)</td>
<td>(w_1^3)</td>
<td>(E_{w_1^0}(E_{w_1^2}(w_1^3)))</td>
</tr>
<tr>
<td>(w_1^0)</td>
<td>(w_1^3)</td>
<td>(w_1^3)</td>
<td>(E_{w_1^0}(E_{w_1^3}(w_1^3)))</td>
</tr>
</tbody>
</table>

The garbled truth table for two inputs is designed in such a way that when it is provided by the sender to the receiver, the receiver can compute only an encryption of the output of the circuit. (Typically, it is the sender who decrypts and uses the output.)

2) Sending the garbled circuit: The sender (could be the originator or another entity that is delegated to use the garbled circuit) sends the garbled truth table representing the garbled circuit to the receiver. Table 3.3 shows an example of a garbled truth table for the garbled circuit (a garbled AND gate) that a sender can send to the receiver. If the sender has input to the garbled circuit it sends also the keys associated to its input (i.e., random values associated to the input of the sender) to the receiver.

3) Oblivious transfer of keys associated with inputs: In Step 1, the originator associated two random values with two circuit inputs. These random values representing inputs
are keys for the garbled circuit. In our simple AND circuit example, there are two circuit inputs and, therefore, two keys.

In this step, the sender sends to the receiver these keys using Oblivious Transfer Protocol (OTP) [EGL85]. Typically, the sender is the originator that holds the keys [CCKM00]. (The originator may also delegate the use of his garbled circuit to another entity.)

For each of its own inputs, the receiver of the garbled circuit selects the key that is appropriate for this input. Then, the receiver uses the keys to execute the garbled circuit. The sender never learns the input values of the receiver.

4) Locally evaluating the garbled circuit: The receiver evaluates the garbled circuit gate by gate. For each gate, the receiver decrypts the output of the gate, using the garbled inputs received in Step 1, and the keys associated with the inputs received in Step 3.

A “very reasonable performance” [LiPi07] is reported for an implementation of the Yao protocol [MNPS2004]. Beaver, Micali and Rogaway [BMR90] present a scheme using garbled circuits to perform multi-party computations.

As shown in Table 3.1, the garbled circuits method can implement both CEF and CEDEF. For CEF, the sender does not provide any input for the garbled circuits; only the receiver provides inputs. For CEDEF, the sender selects his inputs and keys associated with the inputs. Then, he sends to the receiver not only the garbled circuit but also the keys associated with his inputs. The receiver computes the garbled circuit using the keys associated with both the sender’s inputs and receiver’s inputs.
3.3.3. **Hardware-based Methods**

Special hardware for secure computing, e.g., coprocessors, is available from many manufacturers. For example, IBM developed IBM 4758 PCI Cryptographic Coprocessor [IBM09] which includes a secure computing environment for the storage of keys and performing sensitive processing.

3.3.4. **Obfuscation**

Obfuscation refers to hiding program (data and code) in plaintext within code [NaSh08]. The goal of research on program obfuscation is to develop means to scramble code so that it still works, but provably cannot be reverse engineered [Hoh09]. The obfuscation requirements are [Hoh09]:

1) Preserving the program’s functionality, i.e., the obfuscated program should produce the same output as the original one.

2) The obfuscated code can suffer at most a polynomial slowdown compared to the original code.

3) The obfuscated code is a “virtual black box,” i.e., it does not leak any information facilitating reverse engineering.

Barak et al. [BGIR01] prove that it is not possible to have obfuscation that works for any program (based on his definition of obfuscation). We note that results of Barak et al. [BGIR01] do not exclude the possibility that there are obfuscations for some families of functions, such as point functions or deterministic finite automata [KSVZ07].
3.3.5. Privacy Homomorphisms

A privacy homomorphism\(^{18}\) [RAD78, FeHe98, Ferr96] (a.k.a. a homomorphic cryptosystem) is an encryption transformation that allows computations with encrypted data. As a prominent example, the RSA asymmetric-key cryptosystem [Hen08] is a multiplicative homomorphic cryptosystem: given \(E(x)\) and \(E(y)\), where \(E\) is an encryption function, one can calculate \(E(x \times y)\) without knowing \(x\) and \(y\). (The result \(E(x \times y)\) can then be decrypted with the decryption function \(D\) corresponding to \(E\)).

A privacy homomorphism can allow not only multiplications but also additions to be carried out on encrypted data [FeHe98]. As an example, consider the homomorphic encryption scheme of Fig. 3.5 [Gua09]. Fig. 3.6 illustrates using the privacy homomorphism of Fig. 3.5 for additions and multiplications performed on encrypted data.

\[
\begin{align*}
\text{Encryption} \\
y = E(x) = a \mod n \\
\text{where } a = x + r \times p \\
n = p \times q
\end{align*}
\]

\[
\begin{align*}
\text{Decryption} \\
x = D(y) = y \mod p
\end{align*}
\]

Figure 3.5. A homomorphic cryptosystem (cf. [Gua09])

\(^{18}\) A group is a set and an operator that combines any two of the set of elements to form a third element [Wiki2009c]. In general, a homomorphism is a mapping between two groups that respects the group structure [Hom2009]. For example, a mapping \(f\), where \(f(n) = 2n\), is a homomorphism from group \(Z\) to group \(Z\) [Hom2009] since \(f(p+m) = 2(p+m) = 2p+2m = f(p)+f(m)\).
Encrypting $E(x_1 + x_2)$ and $E(x_1 \times x_2)$

Let $x_1 = 5$ and $x_2 = 3$
Let $p = 17$ and $q = 11$, hence $n = p \times q = 187$

Then:
for $r = 8$, $al = x_1 + rxp = 5 + 8 \times 17$ and
$E(x_1) = a_1 \mod n = (5 + 8 \times 17) \mod 187 = 141$
for $r = 20$, $a_2 = x_2 + rxp = 3 + 20 \times 17$ and
$E(x_2) = a_2 \mod n = (3 + 20 \times 17) \mod 187 = 156$

Hence:
$E(x_1 + x_2) = PLUS(E(x_1), E(x_2))$
$= E(x_1) + E(x_2) = 141 + 156 = 297$
$E(x_1 \times x_2) = MULT(E(x_1), E(x_2))$
$= E(x_1) \times E(x_2) = 141 \times 156 = 21,996$

Decrypting $E(x_1 + x_2)$ and $E(x_1 \times x_2)$

$D(E(x_1 + x_2)) = D(297) = 297 \mod 17 = 8$
So indeed: $D(E(x_1 + x_2)) = x_1 + x_2$
$D(E(x_1 \times x_2)) = D(21,996) = 21,996 \mod 17 = 15$
So indeed: $D(E(x_1 \times x_2)) = x_1 \times x_2$

Figure 3.6. Example of computing with a homomorphic cryptosystem (cf. [Gua09])

Table 3.4. Support for different arithmetic and Boolean operations by selected privacy homomorphisms based on Abelian groups

<table>
<thead>
<tr>
<th>Privacy Homomorphism Proposer(s)</th>
<th>$E(x + y)$</th>
<th>$E(x \times y)$</th>
<th>$E(x \text{ or } y)$</th>
<th>$E(x \text{ and } y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ferrer &amp; Herrera-Joancomarti [FeHe98]</td>
<td>yes</td>
<td>yes</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>Goh [Goh07]</td>
<td>yes</td>
<td>yes</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>Kotzanikolau et al. [KBC00]</td>
<td>no</td>
<td>yes</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>Sander &amp; Young [SaYo99]</td>
<td>unknown</td>
<td>unknown</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Another example of computing multiplications and additions with privacy homomorphism is given by Ferrer [Ferr96].

There are two main classes of homomorphic cryptosystems: based on Abelian groups and based on lattices.
1) **Homomorphic Cryptosystems Based on Abelian Groups.** An Abelian group is a cumulative group where the result of applying the group operation to two elements of the group does not depend on the order of the elements [Wiki09d].

Examples of homomorphic cryptosystems based on Abelian groups are: RSA, ElGamal, Goldwasser-Micali, Benaloh, Okamoto-Uchiyama, Paillier, Naccache-Stern, Damgard-Jurik [Hen08], and Forque et al. [FSW03]. Forque et al. provide a privacy homomorphism based on rational numbers, while the other enumerated homomorphic cryptosystems provide privacy homomorphisms based on integers.

Table 3.4 shows support for different arithmetic and Boolean operations by selected privacy homomorphisms based on Abelian groups. Note that a secure privacy homomorphism that allows addition and multiplication modulo 2 automatically supports Boolean operations.19

Ferrer and Herrera-Joancomarti’s privacy homomorphism [FeHe98] is secure against plaintext attacks but not against ciphertext attacks.

Goh’s privacy homomorphism [Goh07] allows only one multiplication. (An output of a multiplication operation must be decrypted before being used for another multiplication operation.)

2) **Homomorphic Cryptosystems Based on Lattices.** A **lattice** is a discrete subgroup of $R^n$ that has infinitely many Z-bases.20 A **subgroup** $H$ is a group where the elements of the subgroup are a subset of a group $G$ [Wiki09e]. For example, the integer

---

19 Let $x, y \in \{0,1\}$. Then $AND(x, y) = xy$, $OR(x, y) = 1 - (1-x)(1-y)$, and $NOT(x) = 1 - x$.

20 Z-base means that the base (or the axis) of the lattice is in group $Z$. In turn, a base $B$ for set $S$ is a set of elements such that elements of $S$ can be written as a combination of elements of $B$ (e.g., $\{0,1\}$ is the base for Boolean numbers).
group $Z$ is a subgroup of the real group $R$; therefore, in particular, any subgroup of $Z^n$ is a lattice [NgSt00].

An example of a homomorphic cryptosystem based on lattices is proposed by Gentry [Gent09a, Gent09b, Gent10]. His homomorphic encryption scheme is secure against plaintext attacks (i.e., an adversary is allowed to ask for encryptions of multiple messages chosen adaptively [KaLi08]). He shows in a recent paper [Gent10] that his scheme supports Boolean operations as well as arithmetic operations. However, his scheme has performance issues.

Johnson et al. [JMSW02] discuss security of homomorphic schemes. They give a set of homomorphic schemes proven to be insecure. For example, the XOR homomorphic schemes are insecure. Ahituv et al. [ALN87] present a successful linear attack on any homomorphic scheme that supports addition with integers, where integers are provided in a binary representation as blocks of 8 or 16 bits.

In our literature search, we did not find a secure homomorphic encryption scheme (either based on Abelian groups or lattices) that supports both algebraic and Boolean operations and has acceptable performance. (Table 3.4 shows that privacy homomorphisms based on Abelian groups are most readily available for algebraic operations.)
3.3.6. **TTP-based Methods**

The TTP-based methods are based on the selection of *trusted third parties* (TTPs) that perform critical computations for interacting entities; all entities provide their inputs to TTPs, and then receive their outputs from TTPs.

Goldreich, Micali and Wigderson [GMW87] describe such a method in the context of a general $n$-player game. To play a game with $n$ players, $n+1$ parties are required where the extra party is a TTP. The TTP communicates privately with each player. At step $t$, the TTP knows the current state of the game $S_t$. Suppose that players take turns in the game so that in step $t$ player $p = t \mod n$ plays. Let $K(p)$ be a function providing the share of knowledge (out of the total knowledge of the game state $S_t$) that player $p$ is entitled to possess. The TTP computes $K(p)$, and sends $K(p)$ to player $p$. (Note that the rest of the knowledge—i.e., “total knowledge” minus $K(p)$—remains unknown to player $p$). Player $p$ uses $K(p)$ to compute his next move $\mu$ and sends $\mu$ to the TTP. Then, the TTP secretly from all players computes the next game state $S_{t+1} = \mu(S_t)$. At the end of the game, the TTP evaluates the final state, and determines the outcome of the game and payoffs for each player.

3.3.7. **Threshold-based Methods**

A threshold-based method of Ben-Or, Goldwasser and Wigderson [BeGW88] provides an unnamed protocol—named by us the BGW protocol—that allows $n$ parties (“players”) for secure collaborative computation of any function $f$ even if some of the parties are adversaries (or malfunction).
The BGW protocol relies in part on the Shamir’s *threshold secret sharing* method [Sham79], which can be described as follows. First, a secret data item $D$ is divided into $n$ pieces $D_1, D_2, \ldots, D_n$. Then, a threshold $t$ is chosen so that: (i) to recover $D$, $t$ or more of arbitrary $D_i$’s are required; (ii) using any $t-1$ or fewer $D_i$’s leaves $D$ completely undetermined (in the sense that all its possible values are equally likely).

The BGW protocol involves $n$ “regular” parties, which calculate only partial function outputs. We distinguish two options for receiving the output: (i) every player receives the output, or (ii) only one player receives the output. In the second option discussed in Ref. [BeGW88], a selected player is named the *dealer* (denoted by us as $DLR$), and is provided the partial function outputs to find out the full results of function computation. In the following we discuss only option (ii).

Let $f$ be a *linear function* of degree $n$ known to each of the $n$ parties, and $t$ be an arbitrary threshold value. Let $P_i$ denote Party $i$, and $x_i$ denote the *secret* input of $P_i$ for $f$. Dealer $DLR$ receives from the $n$ parties the partial outputs of $f$ calculated by the $n$ parties using their respective secret inputs $x_1, x_2, \ldots, x_n$. Let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be distinct non-zero elements in the domain (i.e., members of a set) of $f$. Player $P_i$ is assigned the element $\alpha_i$.

Each party $P_i$ generates a polynomial $h_i$ of degree $t$ (where $t$ is the above threshold value) such that $h_i(0) = x_i$. Each $P_i$ sends to each $P_j$ (from the subset of the other $n-1$ parties) one share $s_{ij} = h_i(\alpha_j)$ of $P_j$’s input. Then, each $P_i$ computes a portion of function $f$ using shares $s_{ij}$ of the input that it has (one) or received from other parties ($n-1$). The $n$ parties send their partial outputs to $DLR$. $DLR$ uses these partial outputs from $n$ parties to compute the complete final output for $f(x_1, x_2, \ldots, x_n)$. So, only $DLR$ “receives” the complete output of $f$. 
The BGW protocol distinguishes the following two types of misbehaving parties among the $n$ parties. First, some of the misbehaving parties are *gossip parties* that follow function $f$ and produce the expected partial outputs of $f$ but in addition try to learn more about secret data $S$ of other parties. $S$ for $P_i$ may include: (i) any input $x_i$ of any other party $P_j$, and (ii) the complete final output of $f(x_1, x_2, \ldots, x_n)$, which should be known only to dealer $DLR$. $P_i$ may attempt to obtain these secret data by disseminating to other $n - 1$ parties their own secret input $x_j$ as well as any other information they gained about $S$, and expecting reciprocity from the other $n - 1$ parties.

Second, other misbehaving parties can be *Byzantine parties* that introduce so called *Byzantine faults* by not using their assigned function $f$ but totally different function, in order to collaborate in learning more about $S$ or even sabotaging computation of $f$ [LaSh82]; as a result, Byzantine parties do not produce the expected output of $f$.

The results for the BGW protocol [BeGW88] indicate that every function can be securely and efficiently computed by $n$ processors (parties). First, if no Byzantine faults occur, no subset of conspiring parties (including gossiping parties) of the size $t < n/2$ (recall that threshold $t$ is equal to the degree of the polynomials used in creating shares of the secret inputs of the $n$ parties) can succeed in obtaining any additional information other than their partial function output. This result is based on the threshold secret sharing method described above.

Second, even if Byzantine faults occur, no subset of conspiring parties (including gossiping parties) of size $t < n/3$ can succeed in either disrupting the computation of $f$ or obtaining any additional information other than their partial function output.
3.3.8. Function Composition

Function composition was proposed by Sander and Tschudin [SaTs98a, SaTs98b] and Lee et al. [LAH04a, LAH04b] as a method that implements secure computing with encrypted functions (CEF).

The function composition method can be described as follows. Let \( f \) be the function to be encrypted, and let an invertible function\(^{21} \) \( g \) be the encryption function. Let function \( h(x) = g(f(x)) \) be the encryption of function \( f \). Alternatively, \( h \) can be shown as the composition of \( f \) and \( g \):

\[
h = g \circ f
\]

Lee et al. [LAH04a] use as an example the functions \( g(x) = x^3+1 \) for encryption of an arbitrary function \( f \) and \( g^{-1}(x) = (x+1)^{1/3} \) for decryption.

As illustrated in Fig. 3.3 (where \( E \) is equivalent to \( g \), and \( D \) is equivalent to \( g^{-1} \)), Bob can send function \( h = g \circ f \) to Alice, who computes \( y = h(x) \) with her input \( x \), and returns \( y \) to Bob. Bob decrypts \( y \) to obtain \( z \) as follows:

\[
z = g^{-1}(y) = g^{-1}(g(f(x)) = f(x)
\]

3.3.9. Hybrid Methods

A hybrid method combines two or more of the simple methods discussed above. For example, a hybrid method could combine coding theory with hardware-based methods.

\(^{21}\) An invertible function \( g \) is a function that has an inverse function \( g^{-1} \) such that \( g^{-1}(g(x)) = x \).
3.3.10. Other Methods

Other methods can be used for secure computing. Methods based on program polymorphism can be one example. “Polymorphism” means here changing the form of an object from plaintext to ciphertext and vice versa.

The basic idea of the program polymorphism method is that at any moment only a small portion of a code to be executed is in plaintext, thus limiting any potential attacker’s access to the full code. In more detail, we have an encrypted code. A portion of this code and its associated data are decrypted before starting a computation. Then, plaintext code and data are used for this partial computation. Finally, after the computation is completed, the code and the associated data are encrypted again.

Other methods for secure computing include: (i) encrypting data and code with keys that become available to the involved parties only under certain environmental conditions (which include timing conditions); and (ii) encrypting data in such a way that a party may decrypt any portion of (or all) encrypted data if it has one key out of the set of decryption keys. Each key can be used to decrypt a different portion of the data.

3.4. Summary

Secure computing is one which does not allow for any unauthorized disclosure of data or functions. We described three approaches for secure computing: secure computing with encrypted data (CED), secure computing with encrypted functions (CEF), and secure computing with encrypted data and encrypted functions (CEDEF).
We summarized several methods proposed in the literature that implement these approaches, namely: coding theory, garbled circuits, hardware-based methods, obfuscation, privacy homomorphisms, TTP-based methods, threshold-based methods, function composition, hybrid methods, and other methods.
4.1. Introduction

Chapter 2 discussed solutions for protecting dissemination of sensitive data. Section 4.2 presents three approaches used by these solutions for protecting disseminated sensitive data when the owner of the sensitive data may not be able to enumerate all entities that are entitled to access the owner’s data. Section 4.2 shows also the limitations of existing approaches. Section 4.3 presents our own proposed approach named the *active bundle scheme*.

We do not discuss the threat model for our active bundle scheme in this chapter since it depends strongly on implementation of the scheme. Chapter 6 presents one of many possible implementations, and discusses the threat model for it.

This chapter is organized as follows. Section 4.2 describes related approaches for protecting confidentiality of disseminated sensitive data. Section 4.3 presents our approach: protecting sensitive data using active bundles. Section 4.4 describes the structure and operations of active bundles. Section 4.5 concludes the chapter.

4.2 Related Approaches for Protecting Confidentiality of Disseminated Sensitive Data

Recall that in Chapter 2 we discussed solutions for protecting confidentiality of disseminated data. In this section we describe three approaches for protecting
confidentiality of disseminated sensitive data and we discuss their limitations. The approaches are: (i) using a Trusted Platform Module (TPM), (ii) using a policy-based identity management encryption scheme, and (iii) using an algorithm-embedded context-aware policy.

4.2.1. Using a Trusted Platform Module (TPM)

Recall that in Subsection 2.3.1 we discussed protecting confidentiality of sensitive data using sticky policies and TPM. It may be wrongly assumed that using a TPM assures secure execution of any code. In fact, it only helps in assuring secure code execution for code trusted by the owner of the TPM-equipped host. It should be remembered that TPM will faithfully execute (“trust”) any software, including software with bugs or malware. Similarly, it will trust software that does not enforce privacy policies correctly. TPM does not prevent its host from tampering with the host’s own policy enforcement programs. In such a way, the host might successfully attack sensitive data.

4.2.2. Using a Privacy Policy-Based Identity Management Encryption Scheme

Recall that in Subsection 2.3.1 we discussed protecting confidentiality of sensitive data using a policy-based identity management encryption scheme ([BaMo05]). Policy-based cryptography allows enforcement of privacy policies. However, it has the following disadvantages:

1) It is applicable only to privacy policy that can be formulated as Boolean expressions. Privacy policies that include obligations (e.g., “data must be destroyed
after 30 days” [KSW03]), context-awareness, and temporal logic (used to control history of data disseminations) are not supported by this scheme because they cannot be expressed as Boolean expressions.

2) It uses TTPs (to provide decryption keys to entities whose credentials satisfy the policies).

4.2.3. Using an Algorithm-embedded Context-aware Policy

A context-aware policy is a set of rules that are activated by the occurrence of contextual events, such as time events. For example, as discussed in Section 2.3.1, Casassa Mont et al. [CaHS02] propose an approach in which a document is encrypted and distributed without the decryption keys needed to access the document; the keys are distributed later, at an appropriate moment.

Context-aware policies in Vanish [GKLL09] (see subsection 2.3.2 for details) and Data Tethers [BER07] (see subsection 2.3.3 for details) are not specified as disclosure rules associated with data but are expressed in the application code. For example, as described in Subsection 2.3.3, data expiration in Vanish is not expressed as a disclosure rule but is embedded in the code of the solution.

4.3. Our Approach: Protecting Sensitive Data Using Active Bundles

As mentioned, we have proposed protecting privacy of sensitive data throughout their lifecycle using the active bundles scheme [BeLi09]. Recall that an active bundle bundles together sensitive data, metadata (including privacy policies) and a VM. Privacy policies include rules for accessing sensitive data. VM includes a policy engine and four protection
mechanisms: integrity self-check, evaporation (to self-destroy endangered portions of data selectively), apoptosis (to self-destroy all data and metadata), and decoy (to mislead suspected attackers with false but harmless information such as “I don’t know” or “No information available”). We describe these protection mechanisms in more detail in Subsection 4.4.2.

The main reason for including VMs in active bundles is to prevent hosts that receive these bundles from accessing sensitive data included in the bundles without enforcing their privacy policies. Using VMs in this way poses two main challenges: (i) how to assure that a visited host executes VM code faithfully and correctly; and (ii) how to implement a VM that protects confidentiality of sensitive data.\textsuperscript{22}

We believe that using a Trusted Platform Module (TPM) is currently the most practical solution to address the first challenge. We assume that an operating system (OS) certified by a TPM executes correctly a VM code.

The second challenge, implementing a VM able to protect active bundles is the main component of this Thesis. The investigated VM implementation approaches include the following:

1) \textit{TTP-based VM implementations.} Our first prototype for the active bundle scheme uses TTP as a security server for the bundle’s VM implementation (details in Chapter 6).

The security server generates encryption/decryption keys, and signing/verification keys. In particular, it provides the active bundle creator (a component of the prototype)

\textsuperscript{22} A VM that protects confidentiality of sensitive data must also protect: (i) integrity of metadata, (ii) confidentiality of the portion of metadata that need to be kept secret, and (iii) integrity of the VM code. (Note that VM code can be public; so, there is no need to protect its confidentiality.) These issues are beyond the scope of this Thesis.
with the encryption and signing key services (to encrypt and sign sensitive data and metadata). It also supplies the bundle’s VM with the decryption keys when its host is allowed to access some or part of the bundle’s data [BeLi09].

2) **VM implementations based on mobile agents.** We considered using mobile agents for implementing VMs for active bundles (details in Chapter 5). We identified critical confidentiality-related properties for the agents. We surveyed literature on mobile agents in search of solutions that protect confidentiality of mobile agents that provide output to visited hosts (which reflects a requirement for the bundle’s VMs). After a critical evaluation of identified candidate solutions, we determined that there is no known solution that satisfies the confidentiality-related agent properties while providing output to a visited host [BeLi10a]. Therefore, there is no available solution that we can be use to implement VM that protects active bundles.

3) **VM implementations based on secure computing.** We investigated the use of secure computing with encrypted data/functions to protect confidentiality of autonomous applications (details in Chapter 7). We proved a set of nonexistence results related to the construction of VMs that implement encrypted autonomous applications [BLGT10]. The results do not mean that there is no solution based on secure computing for this problem. For instance, they do not exclude the existence of VMs that implement domain-specific encrypted autonomous applications (See Section 7.6 for more details).

4) **VM implementations based on program obfuscation.** We initiate an investigation of using program obfuscation to protect confidentiality of autonomous applications (details in Chapter 8). Completion of this work is beyond the scope of this Thesis.
4.4. Structure and Operation of Active Bundles

We discuss in this section the structure and operation of active bundles.

4.4.1. Structure of Active Bundles

Fig. 1.2 shows the structure of an active bundle. An active bundle includes the following components (some were signaled earlier in Section 1.4):

1) *Sensitive data*: It is a digital content that needs to be protected from privacy violations, data leaks, unauthorized dissemination, etc. The digital content can include documents, pieces of code, images, audio or video files.

2) *Metadata*: It describes the active bundle and its privacy policies. The metadata includes (but is not limited to) the following components:

   a. *Provenance metadata*—Including an identifier of the active bundle, identifiers of its creator and owner, the creation date, identifiers of all visited hosts, as well as identifiers of all guardians with their access timestamps and, possibly, their update timestamps if they performed any updates of sensitive data. The history of visits and updates is kept private (it could be used only for forensic investigations by authorities that obtain a court order).

   b. *Integrity check metadata*: It includes the algorithm for checking the integrity of sensitive data as well as a hash calculated by the owner.
c. Privacy policy metadata: It includes access control policies for sensitive data of the bundle. They could use the required minimal host’s trust level for each subset of sensitive data.

d. Dissemination control metadata: It includes dissemination policies specifying who can disseminate the active bundle, and under what conditions (e.g., the active bundle could be disseminated at a specified time, or could be disseminated to a specific group of addressees—such as employees, suppliers, or a board of directors.)

e. Life duration: It specifies a date and time where the sensitive data must disappear.

f. Security metadata: It includes: (i) security server id, specifying the security server used in the process of bundle encryption and decryption (details below); (ii) encryption algorithm used by the VM of the bundle; (iii) trust server id, used to validate the trust level and the role of a host at which the bundle arrived; and (iv) trust level threshold, specifying the minimal trust level required by the VM to enable the active bundle.\(^{23}\)

g. Other application-dependant and context-dependant components: It includes information related to semantics (the application, the context, etc.) of sensitive data of the active bundle.

3) Virtual machine (VM): It manages and controls the program enclosed in an active bundle. The role of a VM is to guarantee the appropriate access control to sensitive

\(^{23}\) If an active bundle is received by a destination host \(H\) with the trust level \(t\), the VM of the bundle must ensure that \(H\) has access to only a portion of data for which \(t\) is a sufficient trust level. If only a portion of data is \textit{not} to be seen by \(H\), evaporation of the data \textit{not} to be shown to \(H\) is performed by the VM. If no bundle data can be seen by \(H\), apoptosis is triggered by the VM.
data of the bundle (for example, disclosing to a guardian only the portion of sensitive data that the guardian is entitled to access). The VM also performs integrity checks for the bundle.

Note that metadata and VM of any active bundle must be constructed in such a way that an attacker who removes the VM from an active bundle must not be able to access its sensitive data. For instance, an attacker must not be able to extract from metadata the address of the security server that stores the decryption keys, and then obtain from the server any decryption key without using the VM of the active bundle. This can be achieved because a security server evaluates the trust level of the visited host, and gives the decryption keys only to hosts that have a trust level sufficient for accessing the sensitive data included in the active bundle. (A host that has a sufficient trust level is assumed to use the VM of an active bundle in order to access the bundle’s sensitive data.)

4.4.2. Operation of Active Bundles

The three basic operations performed by an active bundle are:

1) **Evaporation:** After arriving at a host, an active bundle asks for the host’s trust level. If the hosts’ trust level is sufficient to allow access to all or portion of the bundle’s data, then the bundle’s privacy policy (stored in its metadata) is applied. All data that the host is not allowed to access (as specified in the privacy policy) might be “evaporated.” Evaporation\(^{24}\) of data diminishes the value of data.\(^{25}\)

---

\(^{24}\) The name “evaporation” has been chosen because a host’s “temperature” determines the extent of data “evaporation.”

\(^{25}\)
2) **Apoptosis:** An active bundle may realize that its security or privacy is about to be compromised, e.g., it may discover that its self integrity check fails, or the trust level of its guardian is too low. In response, the bundle may choose to apoptosize (i.e., perform a clean self-destruction, that is, a complete self-destruction that leaves no trace usable for an attacker [LiBh06, Tsch99]). Rules for triggering an apoptosis are included in the privacy policies of the bundle’s metadata. An active bundle might perform other operations, which are beyond the scope of this Thesis.\(^{26}\)

3) **Integrity self-check:** Upon activation of an active bundle at a new guardian, the bundle checks its integrity using the algorithm specified in its metadata. It calculates the hash value of the active bundle, and compares the computed value to the value recorded within the bundle metadata. If the values differ, the bundle performs apoptosis.

### 4.5. Summary

An owner of data may share his data with partners. The partners may in turn disseminate these data further to other entities. If the owner can enumerate all entities that should access his data, he can protect confidentiality of sensitive data using a key-based data encryption approach. Otherwise, he should use an approach based on a privacy

---

\(^{25}\) One way of evaporation is via destroying data that cannot be revealed to the host (data must vanish with no usable trace). Another way of evaporation replaces the original, more valuable, data with approximate data (summary data, averages for ranges, abstracts for reports, etc.), or replaces more current data with older data [LiBh2006].

\(^{26}\) For example, an active bundle may perform *decoy operations* by spreading misleading information at the visited hosts (even if it is not enabled at them) to confuse potential attackers.
policy and its enforcement. Currently, such approaches assume that all hosts receiving the owner’s data are trusted by the owner to enforce the policies associated with these data. We relaxed this assumption. We proposed a new approach based on bundling together sensitive data, metadata, and a VM into a construct named an active bundle. Metadata include privacy policies, and the VM enforces these policies. Active bundles include three protection mechanisms: evaporation, apoptosis, and integrity check.
5.1. Introduction

This chapter (predominantly extracted from [BeLi10a]) investigates mechanisms for protecting confidentiality of mobile agents in cases when they provide output to visited malicious hosts. Recall that an output of a mobile agent\textsuperscript{27} could be either data carried within the mobile agent, or a result of a computation performed by the agent, or both. Recall also that providing output to visited malicious hosts by mobile agents is a requirement for using mobile agents as vehicles for implementing VMs for active bundles.

The chapter first shows results of our search for different solutions proposed for such confidentiality protection. We then evaluate qualitatively these solutions checking if any of them are suitable for using in implementations of VMs for active bundles.

Recall that privacy concerns active entities while confidentiality concerns data (cf. Section 1.1). Based on our definitions of confidentiality and privacy (ibid), “privacy of mobile agents” has a broader semantic scope than “confidentiality of mobile agents” (e.g., it includes the issues of anonymity of agents, their originators or forwarders). In this

\textsuperscript{27} Mobile agents were identified by us in Section 1.5 as a candidate foundation for protecting confidentiality of data and code, which can be used to implement active bundles.
paper, we do not address any issues exceeding the narrower boundaries of agents’ confidentiality.

This introduction starts with a discussion of mobile agents, and their uses and advantages. Second, it presents the motivation for investigating confidentiality of mobile agents. Third, it analyzes mobile agents’ confidentiality. Fourth, it identifies desirable and confidentiality-related properties of mobile agents. Finally, it presents the goals of the chapter and summarizes assumptions of this analysis and evaluation.

5.1.1. Mobile Agents, Their Uses, and Advantages

In this subsection, we introduce mobile agents and mobile-agent architecture; present secure information flow analysis and its relation to protecting confidentiality of mobile agents; and discuss how secure computing approaches can be used to protect mobile agents providing (plaintext) output to visited hosts.

5.1.1.1. Mobile Agents and Mobile-agent Architecture

A mobile agent (a.k.a. a mobile object) is a software object able to perform computations on visited hosts, transport itself from one host to another, and interact with and use capabilities of visited hosts. Being a piece of software, a mobile agent is “passive” and it is “activated” by having its code executed by its host. A mobile agent contains code and carried data, which is data that it brought with it to the host that it is currently visiting.
A mobile agent is created by an originator and disseminated to one or more hosts. A receiving host may disseminate the mobile agent further to other hosts. Alternatively, a mobile agent can self-disseminate to one or more hosts.

Figure 5.1. Client-server vs. mobile-agent application architecture

The most common architecture for applications using distributed data is the client-server architecture. An alternative is to use a mobile agent architecture. Fig. 5.1 illustrates the major difference between these two architectures. In this example, an application that starts on Host S needs data distributed over three hosts: Hosts A, B, and C with data $d_A$, $d_B$, and $d_C$, respectively. In the first, client-server architecture, data must be sent to S from A, B, and C as shown in Fig. 5.1.a. In contrast, in the second architecture, a mobile agent visits A, B, and C to access their data $d_A$, $d_B$, and $d_C$, respectively, as shown in Fig. 5.1.b (the broken line indicates that the mobile agent may—but does not have to—return to its original host S). If the mobile agent is smaller than the total size of needed data, the second architecture saves transmission costs. In addition, the cost of executing a mobile
agent is spread over $A$, $B$, and $C$ rather than being concentrated on $S$, as is the case for the client-server architecture.

The cost-saving potential of using the mobile-agent architecture is counterbalanced by twin threats to an agent’s confidentiality posed by a visited host: threats to the agent’s code, and threats to the agent’s carried data (even if these data are encrypted). In our example from Fig.5.1b, the confidentiality of the agent visiting Hosts $A$, $B$, and $C$ could be compromised if any of these hosts is malicious.\(^{28}\)

5.1.1.2. *Uses and Advantages of Mobile Agent Technology for Applications*

Despite being a concept known for years, mobile agents are still a very active research area [CLX09, CKC09, Out09, RRJO08, TFR06]. In a large measure this is due to advantages offered by them to important new paradigms or technologies.

Some of these advantages are: reducing network load, overcoming network latency, encapsulating protocols, executing asynchronously and autonomously, dynamic adaptability, robustness and fault tolerance [LaOs99], accessing resources locally, and independence from operating systems and hardware [Out09, NIST07].

The paradigms or technologies that benefit from the use of mobile agents and example uses of mobile agents by them include:

---

\(^{28}\) For simplicity, by a *malicious host* we mean here a host on which a mobile agent can be attacked by any attacker, whether it is the host itself, or any entity operating on or via the host. Any unknown or untrusted host is potentially malicious.
1) *Cloud computing.* Cao et al. [CLX09] propose using mobile agents to support management of quality of service (QoS) requested by a consumer and QoS offered by a provider.

2) *Pervasive computing.* Mobile agents can be used for managing disclosures of private patient and staff information through the negotiation of privacy requirements stated by the owner of the private information and required for the requested services (e.g., identity and location of the staff requesting access to a patient’s medical record may be required) [TFR06]. Mobile agents can also be used to facilitate acquiring data from a pervasive sensor network [RRJO08].

3) *Mobile computing.* Disconnectivity problems occur in today’s mobile computing for many reasons. Some are due to malfunctions, some to insufficient coverage by the wireless infrastructure, some to selection of service plans by a given customer. Disconnectivity causes unavailability of remote resources that are essential since mobile devices possess limited resources, such as memory or processing power. Mobile agents facilitate overcoming disconnectivity problems. A mobile agent can perform a computation in the cloud for its application running on a mobile device, and return to the device when the task is accomplished. If a disconnection occurs, the agent simply waits till the connection is restored.

4) *Software engineering.* Mobile agents can be used for the dynamic change of code in running distributed programs. Applications that use mobile agents as components can be remotely maintained by changing appropriate mobile agents without disrupting execution of the applications [CKC09].
5) **Network management.** Mobile agents employed for management of commercial communications speed up locating problems. They are faster than techniques sending alarms through messages [Out09], since they run on devices where the needed resources are available (which reduces the number of messages to exchange with the monitoring system).

Using mobile agents for itinerary planning in routing discovery for ad hoc networks is better than using clients/server communications in terms of energy consumption, network lifetime, and number of hops [Out09].

6) **Smart Grid.** The future Smart Grid will allow energy consumers to buy energy from any power company, store energy locally for future use, produce energy, and sell energy back to power companies. A few companies, some supported by DARPA and the Department of Energy, conduct experiments on the use of mobile agents to implement the Smart Grid [Heyd10, VVDR10a, VVRR10b].

7) **Other widely known paradigms or technologies.** Mobile agents can be used in intrusion detection, distributed data mining, increasing resource availability, discovering and monitoring resources, retrieving information, replicating and collating data, making server configuration backups, as well as information gathering, filtering, and sharing [Wiki10h, Out09].

Implementing mobile agents in various computing systems is facilitated by **frameworks for management of mobile agents.** Examples of such frameworks are: Aglets [Agle10], Voyager [VPCP10], Grasshopper [Gras10], Tryllian [TADK10], JADE [Jade10], SPRINGS [SPM10], Sensorware [Sen10], and MAF [MAF10]. Trillo *et al.* [TIM07] compare performance of some of the above frameworks.
5.1.2. Motivation for Investigating Confidentiality of Mobile Agents

Our interest in mobile agents grew out of the possibility of using them as vehicles for implementing the active bundle scheme. Such use of mobile agents would be facilitated by several similarities exhibited by active bundles and mobile agents, including the following:

1) They have a similar structure. A mobile agent includes data and code. An active bundle contains data, metadata and a virtual machine (code).

2) Active bundles and mobile agents move from a host to another host.

However, active bundles and mobile agents differ, also in the following ways:

1) The basic goals of using a mobile agent include: (i) performing computation at visited hosts, (ii) collecting data from visited hosts, and (iii) providing visited hosts with a mobile agents’ output. In contrast, the goal of using active bundles is to protect data included in an active bundle while providing output to a visited host (if the host is allowed to access them).

2) Mobile agents disseminate themselves using the message push dissemination style. In contrast, active bundles disseminate themselves using message push or message pull, a.k.a. an external repository (cf. Subsection 1.2 for discussion of dissemination styles).

5.1.3. Analysis of Mobile Agents’ Confidentiality

In this subsection we discuss theoretical analysis of mobile agents’ confidentiality, secure information flow analysis, and secure computing approaches.
5.1.3.1. Input/Output Issues for Mobile Agents

Fig. 5.2 illustrates inputs and outputs of a mobile agent performing computations on a visited host. An input is *external* when it is provided by the host, and is *internal* when it is obtained from carried data. Analogously, an output is *external* when it is provided to the host, and is *internal* otherwise (it can be saved as a part of an agent’s carried data).\(^{29}\)

A mobile agent visiting a host may compute with or without any (external) input from the host. Analogously, an agent may give its (external) output to the host or not [Lan98]. This means that a *computation result* obtained by an agent on a visited host can become either: (i) a portion of the agent’s carried data ("internal output"), (ii) an output revealed to the host ("external output"), or (iii) both a portion of the agent’s carried data and an output to the host.

![Diagram of input and output](image)

**Figure 5.2.** External and internal inputs and outputs of a mobile agent performing computations on a visited host

\(^{29}\) From now on, "input/output" denotes "external input/external output" from/to the visited host. In cases when we mean “internal input,” we say so explicitly.
External input and external output are plaintext-equivalent. This means that: (i) the mobile agent either gets a plaintext input from its host or knows how to decrypt the input; (ii) the host receives a plaintext output or knows how to decrypt the output.

When a mobile agent leaves a visited host to move elsewhere, the agent’s input obtained from the host may become a part of the agent’s carried data. (In cases when this input was not incorporated into an agent’s carried data, it was just used for computations at the host and then discarded as not needed any more).

5.1.3.2. Inference of Agent’s Carried Data from Its Code and Output

Consider the situation when a mobile agent’s plaintext code is given to a visited host for execution on the host’s hardware but the agent’s carried data are encrypted. In this case the host cannot see the carried data directly. However, as we assume in this chapter, the host can see the plaintext output of the execution. Knowing both the agent’s code and the output the host can infer values of some agent’s carried data that were used for producing the output.

This means that encrypting an agent’s carried data is not sufficient to prevent inferences disclosing the carried data if the mobile agent’s plaintext code is available to a visited host. The agent’s code needs to be protected from the host as well.

---

30 If the plaintext code of a mobile agent tries to encrypt its output, the host can execute a modified copy of the code, which omits the encryption section of the code.
5.1.3.3. Theoretical Analysis of Mobile Agents’ Confidentiality

Cartrysse and van der Lubbe [CaAv03] present a theoretical analysis of confidentiality of mobile agents. They compared Shannon’s conventional secrecy model with the mobile agent’s secrecy model. Table 5.1 summarizes the results of the comparison. The authors show that it is not possible to achieve secrecy of mobile agents using symmetric encryption.

<table>
<thead>
<tr>
<th>Shannon’s Conventional Secrecy Model</th>
<th>Mobile Agent’s Secrecy Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 type of attackers:</td>
<td>2 types of attackers:</td>
</tr>
<tr>
<td>– during transmission</td>
<td>– during transmission</td>
</tr>
<tr>
<td>2 attacker’s goals:</td>
<td>3 attacker’s goals:</td>
</tr>
<tr>
<td>– extraction of message m</td>
<td>– extraction of task T</td>
</tr>
<tr>
<td>– extraction of key K (the key in symmetric crypto-systems, and the private key in asymmetric cryptosystems)</td>
<td>– extraction of optimal input X</td>
</tr>
<tr>
<td>Allows for symmetric or asymmetric crypto-systems</td>
<td>– extraction of key K (the private key in asymmetric cryptosystems)</td>
</tr>
<tr>
<td>Requires asymmetric crypto-systems</td>
<td></td>
</tr>
</tbody>
</table>

5.1.3.4. Secure Information Flow Analysis

The problem of protecting mobile agent’s carried data when an agent provides output to a visited host is related to secure information flow.\footnote{Information flow from variable $x$ to variable $y$ refers to the transfer of information stored in $x$ to $y$, or to the use of information stored in $x$ to derive information to store in $y$ [Dede1977].} In secure information flow analysis, variables are classified into security levels, indicating sensitivity of the
The so-called non-interference property requires that an entity observing the final values of lower-security-level variables cannot conclude anything about the initial values of higher-security-level variables.

To use the secure information flow analysis in the context of mobile agents, we would first need to classify “variables” from carried data into two classes: lower-security or public information, and higher-security or private information. Then, we would have to assure the non-interference property; that is, prove that a malicious host (“an entity”) observing an agent’s output (“the final values of lower-security-level variables”) cannot conclude anything about the agent’s carried data (“the initial values of higher-security-level variables”). In other words, we would need to assure that an agent’s carried data do not leak.

Unfortunately, the secure information flow analysis does not provide solutions for assuring the non-interference or “no leakage” of carried data for mobile agents providing output to visited hosts. Since the search for and evaluation of possible solutions assuring no leakage of mobile agents’ confidential data are the subject of this paper, secure information flow analysis is not used any more in this survey.

5.1.3.5. Secure Computing Approaches

In contrast to secure information flow analysis, the secure computing approaches are a basis for a number of solutions for protecting confidentiality of mobile agents providing output to visited hosts (the solutions based on the approaches are discussed much later).

The secure computing approaches include: secure computing with encrypted data (CED), secure computing with encrypted functions (CEF), and secure computing with
encrypted data and encrypted functions (CEDEF). More details about the secure computing approaches were provided in Chapter 3.

If we did not allow mobile agents for providing plaintext output to visited hosts, any of the three secure computing approaches could by itself protect mobile agents. In such cases, an approach for protecting a mobile agent’s code could also protect its carried data (e.g., carried data could be split into small portions distributed within code in such a way that by protecting code we also protect data within it). Also, an approach for protecting only a mobile agent’s carried data would be sufficient. The reason is that an attacker given the mobile agent’s plaintext code but no (external) output, cannot infer the plaintext carried data.

However, we do allow mobile agents to provide plaintext output to visited hosts. Under these circumstances, neither CED alone nor CEF alone are sufficient for protecting confidentiality of mobile agents. First, CED keeps agents’ code in plaintext and visible to the visited hosts. It is unable to protect agents’ carried data: knowing both the agent’s code and output, the host can infer values of some agent’s carried data that were used for producing any portion of the output. Second, CEF protects the agent’s code only, and keeps agents’ carried data in plaintext, visible to the visited host. Fortunately, CEDEF is able to protect both agents’ code and agents’ carried data.

Despite the inadequacy of CED and CEF, we consider these partial mobile agents’ confidentiality protection approaches for completeness of the discussion.

The scope of this work is limited to protecting mobile agents’ carried data and code. Hence:
1) We do not consider protecting a mobile agent’s input from its visited host. In fact, we assume that this input, obtained from a visited host, is known to the host.\textsuperscript{32}

2) We do not consider protecting the visited host (including the host-provided input) from a visiting mobile agent.\textsuperscript{33} We can just signal here that Loureiro et al. discuss [LMR00] approaches for protecting hosts from mobile agents, e.g., by using sandboxing.

3) We do not consider the issue of correctness of the input provided to the visiting mobile agent by the visited host. In other words, we assume that the host neither cheats nor makes any mistakes when providing its input.

5.1.4. Desirable and Confidentiality-related Properties of Mobile Agents

Agent autonomy is probably the most important single feature of mobile agents. (The term “non-autonomous agent” is an oxymoron.) Cosmin et al. [CAM07] confirm this conviction by stating: “The most important property on which [the agent researchers] base their discourse is the openness propriety of multi-agent systems, the fact that the agents are self-interested, proactive, know only a local part of the acting world and no central

\textsuperscript{32} To be precise, we need to note that the input might be unknown to the host if it is encrypted by an entity other than the host and only forwarded by the host to the agent. If it is so, this other entity—not the visited host—could infer agents’ carried data if the host shares with it both agent’s plaintext code and output. Ignoring this situation in our explicit considerations does not limit their generality.

\textsuperscript{33} Note that even CEDEF does not protect the visited host’s input from a visiting agent because the agent can include its hosts’ input (either “as is” or after using it in the agent computation) in its carried data and may use it when visiting other hosts.
authority restricts the behaviors of all agents.” Although this citation does not use the word “autonomy” explicitly, we believe that it is exactly what the authors of these words mean.

We distinguish two autonomy-related properties of mobile agents: computation autonomy and data autonomy. They are complemented by other properties of mobile agents, forming together the following set of desirable properties of mobile agents (cf. [Lan98, Out09, TsSo01]):

1) Computation autonomy: A host visited by a mobile agent computes the entire code for the agent by itself. The visited host may request and use data (e.g., input, decryption keys) from other parties while computing the agent’s code.

2) Data autonomy: A host visited by a mobile agent computing code for the agent can use only data that it possesses (e.g., input, decryption keys). The visited host must not receive any data from any other party while computing the agent’s code.

3) Mobility: An agent is able to self-disseminate or be disseminated among hosts [TsSo01].

4) Reactivity: An agent is able to perceive its environment and respond automatically without having to consult a human user (cf. [TsSo01]).

5) Intelligence: An agent is able to learn, reason, and make decisions [TsSo01].

6) Scalability: This is a property of a population of mobile agents, not of a single agent. An agent population can grow without encountering a barrier to growth. It is enabled by agents’ use of the local resources of the visited hosts\footnote{In another interpretation, scalability is a property of a system employing mobile agents; this property is provided or enhanced by the use of mobile agents. For example, since the need to transfer data from hosts} [TsSo01].
Since this chapter focuses on confidentiality of mobile agents, the *confidentiality-related* properties of mobile agents (that is, properties that affect their confidentiality) are most important for our considerations. We believe that mobility, reactivity, intelligence, and scalability do not affect *confidentiality* of mobile agents (rather, they usually impact the *type* of tasks that a mobile agent can perform). In contrast, the two autonomy-related properties of mobile agents turn out to be their confidentiality-related properties as well (cf. [McDo06]).

Due to the importance of confidentiality-related properties of mobile agents, we want to make sure that these properties are well understood, and contrasted with each other.

Consider a host $H$ visited by a mobile agent. Computation autonomy is satisfied only if the entire agent’s code is computed by $H$. In contrast, data autonomy can be satisfied even if any part of the agent’s code is computed by hosts other than $H$.

For instance, the following case satisfies the computation autonomy property. Before computing an agent’s code, the visited host $H$ receives inputs (data) from a set of other hosts. Then, $H$ computes the agent’s code using its own input and the inputs of the other hosts. In contrast, the following case does not satisfy this property. $H$ splits the code of a mobile agent and distributes the code shares to a set of other hosts. Each receiving host computes its code share using its own input, then returns the calculated output to the visited host $H$. $H$ uses the outputs from the other hosts to compute the final output of the agent’s code. Note that in the first case the entire mobile agent is computed by a single
visited host $H$ (using data received from multiple hosts), and, in the second case, the agent’s code is computed by a set of hosts.

Data autonomy is satisfied only if the entire data (e.g., input, decryption key) used by the visited host $H$ in computing the agent’s code is provided by $H$ alone. In contrast, computation autonomy can be satisfied even if a part of data used to compute an agent’s code is provided by hosts other than $H$.

For instance, the following case satisfies data autonomy. A visited host $H$ splits the code of a mobile agent and the associated input data (obtained locally at $H$). It distributes code shares with associated input shares to a set of other hosts. Each receiving host computes its share of the agent’s code using the associated share of input data, then returns the calculated output to the visited host $H$. $H$ uses the partial outputs from the other hosts to compute the final output of the agent’s code. In contrast, the following case does not satisfy this property. A host visited by a mobile agent gets the decryption key for the agent’s code from a trusted third party (TTP). The host uses the key in computing the agent’s code, which results in obtaining the plaintext output of the code. Note that in the first case the other hosts (employed by the visited host $H$) use only data obtained by $H$ locally and provided to them by $H$. In contrast, in the second case, the visited host uses some data that were not in its possession (viz., the decryption key from a third party).

5.1.5. *Chapter Goal and Summary of Assumptions*

This chapter surveys the solutions to the problem of protecting confidentiality of mobile agents visiting malicious hosts in cases when the hosts receive outputs from the
agents. Mobile agents visit malicious hosts to benefit from the local resources owned by the hosts.

The malicious hosts may be authorized to receive from a mobile agent only some data that the agent may contain or produce but not other data. That is, the host is not authorized to use or access these other data. For instance, a host may be authorized to receive the name of a customer from her mobile agent but not her SSN.

Most of the solutions known in the literature protect only some computations performed by mobile agents, namely only the *updates of the state* of a mobile agent while it performs computations on visited hosts [ACCK01] (an example of such a solution is described by Cachin *et al.* [CCKM00]). These solutions work under the assumption that the originator is the only entity that receives the full agent’s output. They are, therefore, unable to protect confidentiality of mobile agents in cases when visited hosts—which could be malicious—also receive some or all output from the agents.

Our goal in this chapter is to qualitatively evaluate solutions for protecting mobile agents known to us from the literature. We want to answer two questions:

1) Are there any available solutions for protecting confidentiality of code and carried data of mobile agents providing output to visited hosts that satisfy the two critical confidentiality-related mobile agent properties: computation autonomy and data autonomy?

2) Are any of these solutions practical?

Answers to these questions are the main contribution of this chapter.

For readers’ convenience, we list the major assumptions used in this chapter, which are discussed above and below. The assumptions already discussed include the following:
A1) Security aspects of mobile agents other than confidentiality are not discussed.

A2) Only confidentiality of mobile agents is considered, not their privacy (which is broader semantically).

A3) A host visited by a mobile agent can see either a part of or the whole plaintext output of the execution of the agent’s code.

A4) A mobile agent’s input is not protected from its visited host. In fact, we assume that this input, obtained from a visited host, is known to the host.

A5) A host visited by a mobile agent is not protected from the agent (including the input provided by the host).

A6) Input received by a mobile agent from a visited host is correct, that is, the host neither cheats nor makes any mistakes when providing its input.

The assumptions discussed below include the following:

A7) Hosts visited by mobile agents (and provided with agents’ output) can be malicious.

A8) We consider only dynamic attacks on confidentiality of mobile agents. Static attacks and dynamic attacks on other aspects of mobile agents (e.g., their integrity) are beyond the scope of this chapter.
5.1.6. Chapter Organization

This chapter is organized as follows. Section 5.2 presents the threat model used in the chapter. Section 5.3 discusses applicability of secure computing approaches for protecting mobile agents, and Section 5.4 discusses applicability of secure computing methods for protecting mobile agents. Section 5.5 presents nine solutions available in the literature, which rely on the secure computing methods. Section 5.6 evaluates the nine solutions qualitatively. Section 5.7 concludes the chapter.

5.2. Threat Model for Mobile Agents Providing Output to Visited Hosts

Mobile agents can perform computations on visited hosts. A visited host interacts with mobile agents to provide input, to perform computations, or to receive mobile agent’s output.

Visited hosts can be adversaries of mobile agents in one of the following ways [LiLi05]:

1) An inquisitive adversary (originally named honest-but-curious [LiLi05]) follows the protocol, but attempts to learn more information than allowed.

2) A hypocritical adversary (originally named weak-honest [LiLi05]) deviates arbitrarily from the protocol as long as such behavior appears honest.

3) A malicious adversary may behave arbitrarily (and does not care for appearances).

This kind of adversary may, for example: (i) refuse to participate in the protocol; (ii) substitute its correct local input with arbitrary data; or (iii) abort the protocol at will.
In our investigation of confidentiality for mobile agents we are interested in cases when a mobile agent provides a visited *malicious* host with its output. Confidentiality of the mobile agent in such a situation may be threatened by the host’s attacks.

We classify *attacks on confidentiality* of mobile agents into two classes: (i) *static attacks*, which are attacks on a mobile agent while it is not performing computations; and (ii) *dynamic attacks*, which are attacks on a mobile agent while it is performing computations.

Static attacks usually take the form of reverse engineering or piracy (cf. [CoTh09]). A *reverse engineering attack* is an attempt to extract the source code of a mobile agent. *Piracy* is an attempt to make an illegal copy of a mobile agent. Static attacks are beyond the scope of this Thesis.

Dynamic attacks on confidentiality of mobile agents usually involve code tampering. We categorize dynamic attacks as: (i) altering the normal sequence of a mobile agent execution (e.g., changing some conditional jump instructions in its program); (ii) replaying the agent execution with a different input; and (iii) altering the state of the agent.

An example of a dynamic attack on confidentiality of a mobile agent that does not tamper with its code (but might tamper with its host’s code) is an attempt to extract information from the memory of its host when the mobile agent is being executed.

We are interested only in dynamic attacks on the *confidentiality* of mobile agents, not dynamic attacks on the *integrity* of mobile agents.
5.3. Applicability of Secure Computing Approaches for Protecting Mobile Agents

As mentioned in Subsection 5.1.3, one of the threats to confidentiality of a mobile agent revealing plaintext output to a visited host is that an attacker (a malicious host) can infer a portion of the agent’s carried data that are used by the agent’s code to compute a portion or the entirety of its output.

In this section, we investigate applicability of secure computing approaches (discussed in Section 3.2) for protecting confidentiality of agent’s code, agent’s carried data, or both from an attacker.

Before proceeding, we need to delineate the following correspondence between the secure computing terminology and the mobile agent terminology:

1) A “function” in secure computing corresponds to “code” of a mobile agent.

2) “Data” in secure computing corresponds to “internal input” or “external input” of a mobile agent (depending on a given solution).

3) A “mobile agent” corresponds to the entity that owns code.

4) A “visited host” corresponds to the entity that provides external input for a given mobile agent.

5.3.1. Applicability of Secure Computing with Encrypted Data (CED) for Protecting Mobile Agents

We discussed secure computing with encrypted data in Subsection 3.2.1. Based on the correspondence between secure computing terminology and the mobile agent terminology, CED protects mobile agents’ carried data but not their code. If we used the mobile agents’
terminology for CED in Fig. 3.2 (in Subsection 3.2.1), Bob would be a mobile agent and Alice a visited host.

In CED, Alice does not have access to Bob’s plaintext data $x$ but only to his encrypted data $E(x)$. Similarly, when CED is used to protect mobile agents, a visited host (“Alice”) does not have access to an agent’s (“Bob’s”) plaintext carried data (“$x$”) but only to his encrypted carried data (“$E(x)$”).

The differences between CED and the mobile agents’ environment are twofold. First, in the former the function $f$ is owned by Alice while in the latter the visited host (“Alice”) does not own the code (“$f$”). Instead, the code is owned by an agent (“Bob”); however, the host (“Alice”) has a full access to the plaintext code (“$f$”) because it must be able to execute this code.

Second, a mobile agent by default is not protected at all (a visited host can access an agent’s plaintext carried data). However, CED makes the assumption that a visited host does not have access to an agent’s plaintext carried data but only to an agent’s encrypted carried data.

5.3.2. Applicability of Secure Computing with Encrypted Functions (CEF) for Protecting Mobile Agents

We discussed secure computing with encrypted functions in Subsection 3.2.2. Recall that the term function used in the literature on secure computing corresponds to the term code in the context of mobile agents.

\[\text{For brevity, instead of saying corresponding to } X \text{ we say just “}X” \text{ (in quotes).}\]
Based on the correspondence between secure computing terminology and the mobile agent terminology, CEF protects agents’ code but not their carried data. If we used mobile agents’ terminology for CEF in Fig. 3.3 (in Subsection 3.2.2), Bob would be a mobile agent, and Alice a visited host.

In CEF, Alice does not have access to Bob’s plaintext function \( f \) but only to his encrypted function \( E(f) \). Similarly, when CEF is used to protect mobile agents, a visited host (“Alice”) does not have access to an agent’s (“Bob’s”) plaintext code (“\( f \)”) but only to his encrypted code (“\( E(f) \)”).

The differences between CEF and the mobile agents’ environment are threefold. First, in CEF Alice’s input is protected from Bob while in the mobile agents’ environment an agent (“Bob”) may include the visited host’s (“Alice”) input in its carried data and may use it in its computation on other hosts that it may visit.

Second, in CEF Bob does not provide any data for Alice’s computation while in the mobile agents’ environment an agent (“Bob”) may provide some of its carried data as a part of input used by the visited host (“Alice”) to compute output of the agent’s code.

Third, a mobile agent by default is not protected at all (a visited host can access an agent’s plaintext code). However, CEF makes the assumption that a visited host does not have access to an agent’s plaintext code but only to an agent’s encrypted code.

5.3.3. Applicability of Secure Computing with Encrypted Data and Encrypted Functions (CEDEF) for Protecting Mobile Agents

We discussed secure computing with encrypted data and encrypted functions in Subsection 3.2.3. Based on the correspondence between secure computing terminology
and the mobile agent terminology, CEDEF protects both mobile agents’ code and carried data. If we used the mobile agents’ terminology for CEDEF in Fig. 3.4 (in Subsection 3.2.3), Bob would be a mobile agent, and Alice a visited host.

In CEDEF, Alice does not have access to Bob’s plaintext data \( y \) and plaintext function \( f \) but only to his encrypted data \( E(y) \) and encrypted function \( E(f) \). Similarly, when CEDEF is used to protect mobile agents, a visited host (“Alice”) does not have access to an agent’s (“Bob’s”) plaintext data (“\( y \)”) and plaintext code (“\( f \)”) but only to his encrypted data (“\( E(y) \)”) and encrypted code (“\( E(f) \)”).

The differences between CEDEF and the mobile agents’ environment are twofold. First, in CEDEF Alice’s input is protected from Bob while in the latter an agent (“Bob”) may include the visited host’s (“Alice”) input in its carried data and may use it in its computation on other hosts that it may visit.

Second, a mobile agent by default is not protected at all (a visited host can access agent’s plaintext code and plaintext data). However, CEDEF makes assumption that a visited host does not have access to an agent’s plaintext code and plaintext data but only to an agent’s encrypted code and encrypted data.
5.4. Applicability of Secure Computing Methods to Protecting Mobile Agents

Section 3.3 describes ten methods known in the literature that protect confidentiality of computing using secure computing with encrypted data or/and encrypted functions. This section discusses applicability of these methods for protecting mobile agents providing output to visited malicious hosts.

5.4.1. Coding Theory

In the mobile agent context, coding theory can be used to protect an agent’s code/function $f$ represented by a matrix $M$. Coding theory first transforms $M$ to $M'$, where $M' = M \cdot P$ ($P$ is a random matrix). Then, it includes $M'$ within a mobile agent. Next, the host that receives the agent computes $M'$ to produce its output $y'$. The mobile agent returns to the originator with $y'$. The originator uses the inverse of $P$, $P^{-1}$, to compute $y$ from $y'$.

5.4.2. Garbled Circuits

An example of using garbled circuits for protecting confidentiality of mobile agents follows. The originator creates a garbled circuit that represents the agent’s function and includes it in the agent. Then, the originator sends the agent to a destination host. The host communicates with the originator to obtain the keys associated with the originator’s inputs for the agent. Next, the host computes the garbled circuits using the keys to obtain the output of the garbled circuits (which is the encrypted output of the agent). After that, the host sends the encrypted output to the originator who decrypts it to obtain the plaintext output for the mobile agent.
5.4.3. **Hardware-based Methods**

An example of using a hardware-based method for protecting confidentiality of mobile agents is as follows. A mobile agent is encrypted using the public key of a destination host. Then, the originator sends the agent to the destination host that has a trusted coprocessor. Upon activating the agent, its code is transferred to the trusted coprocessor for decryption, using a private key stored in its memory. Then, the coprocessor computes the output for the mobile agent using the input provided by the host.

5.4.4. **Obfuscation**

Recall that Barak *et al.* [BGIR01] do not exclude the possibility that there are obfuscations for some families of functions [KSVZ07]. For this reason we include obfuscation among potentially useful methods for protecting privacy of mobile agents.

An example of using obfuscation for protecting confidentiality of mobile agents is as follows. The originator obfuscates code and data of an agent and sends it to a destination host. The destination host executes the obfuscated agent providing its input. The agent can give the host the plaintext output of the computation without endangering its confidentiality.

Note that currently, there are no known obfuscation methods for mobile agents based on secure computing. However, there are methods for program obfuscation based on some properties of programming languages [CoTh09], or based on cryptographic constructs [Hoh09, KSVZ07] (though the latter have limited use).
Majumdar and Thomborsom [Math05] propose obfuscating mobile agents using *opaque predicates*. Udupa et al. [UDM05] show the efficiency limitations of such a code obfuscation technique. In particular, they developed a technique for deobfuscation (extraction of the original program from an obfuscated program). It eliminates 73% of code introduced by obfuscation using inter-procedure data flows, and 78% of code introduced by obfuscation using indirect memory accesses. (Inter-procedure data flows and indirect memory accesses are two programming language constructs that can be used to construct opaque predicates.)

### 5.4.5. Privacy Homomorphisms

An example of using a privacy homomorphism for protecting confidentiality of mobile agents is as follows. The originator encrypts data of a mobile agent and sends them to a destination host. The destination host gives the agent its input and executes it. The agent returns to the originator who gets the encrypted output from the mobile agent, and decrypts it to obtain the plaintext output of the computation.

Note that Sander and Tschudin [SaTs98a, SaTs98b] are among the authors who state in their papers that if there were a secure homomorphic scheme, it could be used to secure computations of mobile agents. It should be noted that securing computations of mobile agents was “previously believed to be impossible without trusted hardware” [SaTs98b].

---

36 An *opaque predicate* is a conditional expression whose value is known to the obfuscator (i.e., the party that creates obfuscation of a program) but is difficult to deduce by an attacker.

37 *Inter-procedure data flow* refers to flow of data via parameters function calls.

38 *Indirect memory access* means accessing memory via pointers rather than regular variable names.
5.4.6. **TTP-based Methods**

An example of using a trust-based method for protecting confidentiality of mobile agents is as follows. The originator of a mobile agent and the destination host agree on a TTP. The originator sends the agent, and the host sends the data to the TTP. The TTP executes the agent and returns the output of the agent’s code to the host and the originator.

5.4.7. **Threshold-based Methods**

An example of using a threshold-based method for protecting confidentiality of mobile agents is as follows. An originator $O$ of a program generates a decryption key and splits it into $t$ shares using the *threshold secret sharing* method [Sham79]. $O$ sends $t$ shares to $t$ different hosts. Next, $O$ splits the program that $O$ wants to be executed among $t$ mobile agents, and sends one agent to each of the $t$ hosts. Each host executes “its” agent using host’s input, and obtains its agent’s partial encrypted output. Each of the $t$ hosts decrypts its partial output using its key share, and sends its share of the decrypted output to $O$. Originator $O$ combines the shares to obtain the complete output of the original program.

5.4.8. **Function Composition**

An example of using function composition for protecting confidentiality of mobile agents is as follows. The originator transforms function $f$, implemented by the code of a mobile agent, using encryption function $g$. The code of the agent is replaced by code that implements the composite function $h = g \circ f$. The originator sends the agent to a destination host. The host executes the agent using the host’s input, and obtains the agent’s
encrypted output. Then, the host sends encrypted output to the originator. The originator “decrypts” the output using $g^j$.

5.4.9. Hybrid Methods

Solutions for protection of data and/or code for mobile agents can be based not only on using individually the simple methods listed above; they can also use hybrid combinations of these simple methods. For example, we later present a solution that combines coding theory with hardware-based methods (cf. Solution S4 in Subsection 5.5.4).

5.4.10. Other Methods

Recall that other methods for secure computing include: (i) encrypting data and code with keys that become available to the involved parties only under certain environmental conditions (which include timing conditions); (ii) encrypting data in such a way that a party may decrypt any portion of (or all) encrypted data if it has one key out of the set of decryption keys; each key can be used to decrypt a different portion of the data.

Code or data protection provided by such “other” methods can be a candidate for assuring mobile agent confidentiality.
Figure 5.3. A taxonomy of secure computing solutions for protecting mobile agents
Table 5.2. Solutions and secure computing methods and approaches used by them.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Secure Computing Method(s) Used</th>
<th>Secure Computing Approach Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: Protecting mobile agents using the E-E-D property</td>
<td>Other methods (in Protecting Data)</td>
<td>—</td>
</tr>
<tr>
<td>S2: Protecting mobile agents using hardware-based secure computing</td>
<td>Hardware-based methods</td>
<td>CEDEF</td>
</tr>
<tr>
<td>S3: Protecting mobile agents using obfuscation</td>
<td>Obfuscation</td>
<td>—</td>
</tr>
<tr>
<td>S4: Protecting mobile agents using coding theory and hardware-based methods</td>
<td>Coding theory</td>
<td>CEF</td>
</tr>
<tr>
<td></td>
<td>Hardware-based methods</td>
<td>CED</td>
</tr>
<tr>
<td>S5: Protecting mobile agents using garbled circuits and TTPs</td>
<td>Garbled circuits</td>
<td>CEDEF</td>
</tr>
<tr>
<td></td>
<td>TTP-based methods</td>
<td>—</td>
</tr>
<tr>
<td>S6: Protecting mobile agents using garbled circuits and threshold-based multi-party computing</td>
<td>Garbled circuits</td>
<td>CEDEF</td>
</tr>
<tr>
<td></td>
<td>Threshold-based methods</td>
<td>—</td>
</tr>
<tr>
<td>S7: Protecting mobile agents using privacy homomorphisms and function composition</td>
<td>Privacy homomorphisms</td>
<td>CED</td>
</tr>
<tr>
<td></td>
<td>Function composition</td>
<td>CEF</td>
</tr>
<tr>
<td>S8: Protecting clueless mobile agents using environmental conditions</td>
<td>Other methods (in Protecting Data and Functions)</td>
<td>—</td>
</tr>
<tr>
<td>S9: Protecting mobile agents using polymorphism</td>
<td>Other methods (in Protecting Data and Functions)</td>
<td>—</td>
</tr>
</tbody>
</table>

Notes:
1. Multiple entries in the Secure Computing Method(s) Used column indicate that a hybrid method is used by the corresponding solution.
2. A dash in the rightmost column indicates that a given method does not use any of the three secure computing approaches.

5.5. Nine Available Solutions for Protecting Mobile Agents Providing Output to Visited Malicious Hosts

In this section, we present nine solutions identified by us available in the literature that seemed promising for protecting confidentiality of mobile agents in cases when they provide output to visited malicious host.
Fig. 5.3 shows the taxonomy of the solutions, relating the solutions to approaches and methods presented in Chapter 3. The top level in Fig. 5.3 includes root/header of the taxonomy. Level 2 shows approaches, Levels 3 and 4—methods, and Level 5—individual solutions. In more detail, approaches at Level 2 are classified as: (i) protecting data (using CED or not); (ii) protecting functions (in general, the approaches might be using CEF or not; however, both methods for protecting functions shown in Fig. 5.3 use CEF); and (iii) protecting both data and functions (using CEDEF or not). At Level 3, we see methods that can be used for each approach from Level 2. Level 4 shows simple methods that are components for Hybrid Methods from Level 3. (Level 4 is empty for simple, non-hybrid methods.) Level 5 shows the nine prospective solutions identified by us, connected to the methods they use.

Table 5.2 identifies methods and approaches used by each prospective solution. Only one solution, S1, is not a complete solution in the sense that it does not protect both code and carried data of mobile agents. S1 is able to protect only the agent’s carried data but not its code.\(^{39}\) None of the nine solutions protects only a mobile agent’s code. The remaining eight solutions are complete; protecting both the agent’s code and agent’s carried data.

### 5.5.1. Solution S1: Protecting Mobile Agents Using the E-E-D Property

Cartrysse and van der Lubbe [CaAv03] present a set of techniques for protecting confidentiality and integrity of disseminated mobile agents, as well as a theoretical

\(^{39}\) Recall that in this Thesis the terms functions and code are synonyms.
analysis of security of mobile agents based on Shannon’s communication theory of secret systems [Shan49]. The solution has two variants:

1) A mobile agent that is disseminated to a known set of hosts, say \( n \) hosts is copied \( n \) times, with each copy protected by the appropriate host’s key.

2) A mobile agent that is disseminated to a known set of hosts uses an encryption scheme that has the so called E-E-D property (where E refers to encryption and D refers to decryption in a chain of asymmetric key encryptions and decryptions).

The second, E-E-D-based, variant of Solution S1 can be summarized as follows. The originator \( O \) of a mobile agent encrypts (denoted by the first \( E \)) the mobile agent with the agent’s public key. Next—when the mobile agent must be transferred by \( O \) to a partner \( P_i \) — \( O \) encrypts (denoted by the second \( E \)) the mobile agent using \( P_i \)’s public key. Then, \( O \) decrypts (denoted by \( D \)) the encryption using \( O \)’s own private key. In turn, when \( P_i \) wants to transfer the mobile agent to another host \( P_j \), then: (i) \( P_i \) encrypts the mobile agent with the public key of \( P_j \); (ii) decrypts it with the \( P_i \)’s own private key; and (iii) forwards the result to \( P_j \).

The following property assures that the E-E-D scheme works:

\[
D_{PRI(i)} ( E_{PUB(j)} ( E_{PUB(i)} (m) ) ) = E_{PUB(j)} (m)
\]

where \( PRI(k) \) and \( PUB(k) \) are the private and the public keys for \( P_k \), respectively.

The E-E-D property guarantees that a mobile agent can be disseminated in an encrypted form even to hosts that the originator does not know (however, these destination hosts must be trusted by the source hosts that forward the agents to them). Observe that a mobile agent stays encrypted after it leaves its originator (but the encryption changes as the agent is being disseminated).
The E-E-D-based variant of Solution S1 assures that a mobile agent is stored in encrypted form on a visited host. It also assures that a mobile agent is sent from a source to a destination in an encrypted form. It assumes that the host follows the sequence of encryptions/decryptions described above. A malicious host can cheat and decrypt data using its own private key to get access to data.

The solution has two serious drawbacks. First, it is the only one that does not protect agents’ code. Second, it has a scalability problem since it is based on assigning a key to each user who wants to use a mobile agent. If \( n \) copies of code and data for \( n \) users are included in the same mobile agent, the agent must be encrypted \( n \) times.

5.5.2. Solution S2: Protecting Mobile Agents Using Hardware-based Secure Computing

Yee [Yee99] proposes using a trusted coprocessor (i.e., tamper-proof hardware) and trusted Java agents as an environment to run mobile agents visiting potentially malicious hosts. The solution is based on having a mobile agent framework running in the trusted environment.

The idea for Solution S2 is that mobile agents are executed by an unmodified Java interpreter using a trusted co-processor. Before sending a mobile agent, the originator encrypts the agent using the public key of the destination host. The associated private key is stored in the trusted coprocessor of the destination host (not on the destination host itself). To execute an agent, the coprocessor first decrypts the agent using the private key, and then runs the agent’s code (after making sure that the Java Virtual Machine was not tampered with).
Two drawbacks for solutions using hardware-based secure computing are: (i) having a trusted coprocessor on all hosts that a mobile agent might visit is expensive; and (ii) the assumption that hardware on any host visited by a mobile agent is trusted by the originator of the agent is very strong; it either restricts the range of hosts that the agent is allowed to visit, or can be easily invalidated by the mobile agent unexpectedly visiting a host without a coprocessor trusted by the agent’s originator.

5.5.3. Solution S3: Protecting Mobile Agents Using Obfuscation

The goal of research on program obfuscation is to develop means to scramble code so that it still works, but provably cannot be reverse engineered [Hoh09].

Solution S3 could use any of the several implementations for program obfuscation in an untrusted execution environment. Chow et al. [CEJO02] present an implementation using the DES block cipher, but it was broken by Jacob et al. [JBF02]. Link and Neuman [LiNe05] propose an improved implementation of the obfuscation method of Chow et al., but their solution was also broken [GMQ07, WMGP07].

Chow et al. [CEJO02b] propose another implementation for program obfuscation, which uses the AES cipher, but also this implementation was broken [BGE05].

Wyseur [Wys09] provides a thorough investigation of existing practical solutions for white-box cryptography and their security. He concludes that existing practical obfuscations were broken because they use AES or DES, which are not strong one-way functions [Wys09].

The drawback of this solution is that it is difficult to obfuscate arbitrary code. Barak et al. [BGIR01] prove that it is not possible to have obfuscation that works for an arbitrary
program. In particular it is believed that obfuscating deterministic programs is impossible [BGIR01], since it is possible to trace the execution of a program and extract its flow of control. Deterministic programs are a subclass of arbitrary code. Other subclasses include probabilistic programs.

5.5.4. Solution S4: Protecting Mobile Agents Using Coding Theory and Hardware-based Methods

Loureiro [Lour01] protects confidentiality of computations and data for mobile agents visiting possibly malicious hosts. The algorithm uses the coding theory to protect confidentiality of computations, and uses a tamper-proof hardware to protect confidentiality of data. Security of the proposed algorithm is based on security characteristics of the McEliece asymmetric key cryptosystem (in which the private and public keys are large matrices) [BMT78].

The algorithm uses matrix $M$ to represent an arbitrary function $f$. To generate $M$, one develops a set of linear equations that map any input set $\{x_i\}$ of function $f$ to its output $\{f(x_i)\}$, i.e., $f(x_i) = M \times x_i$.

Solution S4 uses this coding-theory-based algorithm for protecting code, not data. In S4, matrix $M$ is transformed to matrix $M'$. $M'$ is then included in a mobile agent, accessible to the agent’s visited hosts.

The algorithm, shown in Fig. 5.4, is as follows: Alice (“a mobile agent“) has a function $f$ to be evaluated by Bob (“a visited host”) on his input $x$. Alice hides her matrix $M$ for the function $f$ as $M' = M \times G \times P$, where $G$ is a generator matrix and $P$ is a random permutation matrix. Alice sends $M'$ to Bob who evaluates $M'$ for his input $x$ and returns
\[ y' = M'x + e, \text{ where } e \text{ is a random error vector.} \]

Alice decrypts \( y' \) using the inverse of matrix \( G \), the inverse of matrix \( P \), and matrix \( e \) (\( e \) is a secret shared between the two parties). The function evaluation result is \( y = Mx \).

The protocol guarantees that Alice obtains the correct result, while preventing disclosure of \( f \) to Bob and disclosure of \( x \) to Alice.

The solution has the same drawbacks as Solution S2, namely: (i) having a trusted coprocessor on all hosts that a mobile agent might visit is expensive; and (ii) the assumption that hardware on any host visited by a mobile agent is trusted by the originator of the agent is very strong.

**5.5.5. Solution S5: Protecting Mobile Agents Using Garbled Circuits and TTPs**

Algesheimer et al. [ACCK01] maintain that existing solutions for computing with encrypted function are limited to cases when hosts visited by a mobile agent may
participate in its computing, but do not learn anything from the output of the agent. They claim that their solution, unlike the previous ones, allows a host to get an agent’s output.

The solution, using a Yao’s circuit [Yao86] and a trusted third party (TTP), assumes that the visited host provides correct input. However, the host could still attack the mobile agent in other ways, e.g., by being inquisitive [KiSc06]. Fortunately, Yao’s circuit can be used to protect against inquisitive adversaries [KiSc06].

Li and Li [LiLi05] indicate that solutions based on Yao’s circuit can be improved—by using commitment protocols—to provide protection also against hosts providing incorrect input.

The idea for Solution S5 is as follows. An originator $O$ creates a garbled circuit for $O$’s function $f$. $O$ also generates a set of keys and associates them with possible inputs (see Subsection 3.3.2 for details about garbled circuits). Then, $O$ includes the garbled circuit in a mobile agent as its code, and sends the agent to a destination host $H$. $O$ also sends the keys to a TTP. Before computing the mobile agent, host $H$ communicates with the TTP to obtain the keys associated with inputs of the garbled circuit. After computing the agent, $H$ sends the encrypted output to the originator $O$. $O$ decrypts the encrypted output to obtain the plaintext output.

The proposed scheme assumes [ACCK01] that: (i) the TTP does not collude with any hosts against the originator; and (ii) the TTP does not collude with the originator against any host.

Some of the drawbacks of the solution are: (i) it is a centralized solution (since the TTP is centralized), so—by its very nature—not fault-tolerant; and (ii) it assumes that all hosts and originators trust one entity, TTP. This is a very strong requirement, difficult to satisfy
for a system of a realistic size (in which some participants will distrust any given TTP) (cf. [TaXu03]).

5.5.6. Solution S6: Protecting Mobile Agents Using Garbled Circuits and Threshold-based Multi-party Computing

Tate and Xu [TaXu03] propose a multi-party solution for securing a mobile agent on malicious hosts in cases when the hosts receive an output from the agent. The authors based their work on Algesheimer’s results [ACCK01]. The scheme uses Yao’s circuit [Yao86] but instead of relying on a TTP to perform some computations, the threshold decryption key technique is used.

The multi-party computing in Solution S6 involves $n$ hosts, including $s$ hosts that might collude. An originator $O$ of a mobile agent first generates a public encryption key $PK$ and a secret decryption key $SK$ for the agent. Second, $O$ creates a garbled circuit $C$ for $O$’s function $f$. Recall that each row of the garbled truth table for $C$ is associated with one gate. A row includes two keys $w_1$ and $w_2$ associated with two possible inputs for a gate (one input for each wire of the gate) such that $w_1$ and $w_2$ could be used to decrypt the output of the gate for the associated inputs. Third, $O$ encrypts all keys associated with possible inputs using $PK$. Fourth, $O$ splits the garbled circuit into $n$ “partial” garbled circuits or “GC shares.” Fifth, $O$ sends to Host $i$ ($i = 1, 2, \ldots, n$) the $i$-th mobile agent (including the $i$-th GC share as its code), and $SK_i$—the $i$-th share of the secret decryption key $SK$.

Host $i$ communicates with $t$ other hosts to get from them $t$ decryption key shares for its inputs (that is, for the inputs for its GC share). The host combines the decryption key
shares to obtain the keys associated with all its inputs, and uses the keys to compute the plaintext output for its mobile agent.

Finally, the host sends the mobile agent including the encrypted output to the originator. The originator decrypts this output using $SK$, and then combines the outputs of the $n$ agents to obtain the final input.

The threshold $t$ is a security parameter such that a mobile agent is protected when $\text{ceiling}\left(\frac{n+s}{2}\right) \leq t$. Given $t$, any set of fewer than $s$ malicious hosts cannot gain unauthorized access to protected computation output. In other words, no set of fewer than $s$ colluding hosts can decrypt together any of the outputs of mobile agents.

The advantages of the solution [TaXu03] are that: (i) it does not require a TTP; (ii) it provides some protection against colluding hosts; (iii) it is fault-tolerant; and (iv) it supports parallelism, since $n$ hosts can compute their partial agent’s output decryptions in parallel.

The main disadvantage of the solution is the high cost of communication among the hosts involved in computing the partial agent’s outputs, and the fact that $t$ hosts need to participate in the decryption of the agent’s output.

5.5.7. Solution S7: Protecting Mobile Agents Using Privacy Homomorphisms and Function Composition

Sander and Tschudin [SaTs98a, SaTs98b] were among the pioneers in research on the use of privacy homomorphisms and function composition for protecting data and computations for mobile agents visiting hosts (including malicious hosts). To the best of our knowledge, they were the first ones who talked about the use of privacy
homomorphisms for protection of software or mobile agents (others provided homomorphic schemes, but did not even mention such uses).

A two-party/single-interaction algorithm with non-interactive function computing is an algorithm in which there is no communication with any other party during computing of the function; however, a single interaction occurs in the algorithm either before or after computing the function.

Consider, for example, the simple two-party/single-interaction algorithm with a non-interactive function computing using an encrypted function [SaTs98b] shown in Fig. 5.5.

Solution S7 uses, as follows, privacy homomorphisms to implement computing with encrypted data. Alice encrypts \( f \), and creates a program \( P(E(f)) \) which implements \( E(f) \) (where a program here is a mobile agent). She then sends \( P(E(f)) \) to Bob. Bob computes \( P(E(f))(x) \), and sends this result to Alice. Alice decrypts \( P(E(f))(x) \) and obtains \( f(x) \).

Note that computing function \( P(E(f))(x) \) by Bob in Fig. 5.5 is non-interactive since Bob does not communicate with Alice during computing of the function \( P(E(f))(x) \); Bob communicates with Alice only before and after this computing.

Sander and Tschudin [SaTs98a, SaTs98b] discuss implementing the algorithm of Fig. 5.5 using privacy homomorphisms (e.g., using privacy homomorphisms to compute polynomial functions).

Lee et al. [LAH04a, LAH04b] propose an extension to Sander and Tschudin’s work [SaTs98a, SaTs98b]. The proposed solution uses privacy homomorphism to protect data privacy, and uses function composition to protect function privacy.
To protect privacy of data, the authors propose a new cryptosystem named MMH (Mixed Multiplicative Homomorphic) cryptosystem which supports privacy homomorphism. Fig. 3.5 (in Chapter 3) shows examples of encryption and decryption functions for MMH, and Fig. 3.6 (in Chapter 3) shows an example of using the privacy-homomorphic property of MMH [Gua09].

To protect privacy of functions to be computed on malicious hosts, the authors propose using function composition.

Solution S7 has two main drawbacks:

1) S7 can only be used for polynomial functions.

2) S7 is not secure if a univariate polynomial function is used in a function composition method to hide the second function. The reason is that a malicious host can decompose the composed function using the lattice reduction technique [LLL82], thus finding out the plaintext agent’s function (which is a problem since the host should be able to see only the encrypted agent’s function).
5.5.8. Solution S8: Protecting Clueless Mobile Agents Using Environmental Conditions

Riordan and Schneider [RS98] present their clueless agents scheme that uses environmental variables to generate keys used by cryptographic primitives. For instance, the keys are generated to decrypt a mobile agent when a specific environmental timing condition occurs (time is an example of an environmental variable that can be used to decrypt a mobile agent). The idea is that a mobile agent must obtain or get a set of environmental data for key generation, maybe from a web site, a visited host, or a message.

![Sequence diagram for encryption of a message using the clueless agent scheme](image)

**Figure 5.6. Sequence diagram for encryption of a message using the clueless agent scheme**

Fig. 5.6 and Fig. 5.7 illustrate the main components of the clueless agent scheme.

The idea for Solution S8 is as follows. As Fig. 5.6 shows, to encrypt a plaintext message \( m \), the originator of the agent gets a particular possible output \( N \) of program \( P \), calculates its hash function \( H(N) \), and sends program \( P \) and \( H(N) \) to the key server. The key server returns to the originator: (i) the encryption key \( K \), generated as the hash value \( K = H( S || P || H(N) ) \); and (ii) the encrypted program \( E_S(P) \), where \( S \) is a secret key of the
key server, which stores secret keys. The originator generates the encrypted message $C$ by encrypting the plaintext message $m$ with the key $K$, and sends $C$ and $E_S(P)$ to the agent.

As Fig. 5.7 shows, to decrypt $C$, the agent sends $E_S(P)$ to the key server. The key server decrypts program $P$: $P = D_S(E_S(P))$, runs $P$ to get its output $N'$, and calculates hash value $M = H(N')$. The server sends $K = H(S \mid P \mid M)$ to the agent. The agent tries to use the received key $K$ to decrypt the message. The agent succeeds only when $M = H(N)$.

In practice (for good hash functions), this happens only if $N' = N$, where $N'$ is the output of program $P$ obtained by the key server after the agent’s decryption request (cf. Fig. 5.7), and $N$ is the output of $P$ obtained by the programmer at the beginning of the algorithm (cf. Fig. 5.6).

![Figure 5.7. Sequence diagram for decryption of a message using the clueless agent scheme](image)

We have identified two drawbacks and found in the literature two more drawbacks of Solution S8:

1) Once a mobile agent is decrypted while visiting a host, its behavior could be changed by the host.

2) Once a mobile agent is decrypted while visiting a host, it becomes plaintext, and can be disseminated to other hosts as a plaintext agent.
3) The entry point for a mobile agent’s code is a plaintext code, including: (i) a test that must be passed to decrypt the mobile agent; and (ii) the decryption program. Such code could be changed by an unauthorized visited host (e.g., the code of the activation condition could be changed) [McDo06].

4) Some hosts visited by mobile agents may be unable to execute dynamic code [McDo06]. Such hosts differentiate memory zones using a flag that indicates whether the memory zone supports only execute operations or read/write operations. These hosts do not permit a mobile agent to be decrypted in a memory zone, and later executed in the *same* memory zone. The reason is that a decrypted mobile agent—considered data—must be kept in a *data* memory zone, while execution of the mobile agent must be done in a *program-code* memory zone.

5.5.9. **Solution S9: Protecting Mobile Agents Using Polymorphism**

Cappaert and Preneel [CPAM08] present three techniques for protecting disseminated data using polymorphism: bulk decryption, on-demand decryption, and a combination of both techniques. Recall that “polymorphism” means here changing the form of an object from plaintext to ciphertext and *vice versa*. A program (e.g., a mobile agent) that uses polymorphism must include an entry point for a decryption function.

Solution S9 uses one of the following three techniques. In the *bulk decryption* technique, the entire program must be decrypted before it can be executed. The decryption may use a key that is generated on the fly (e.g., using the size of the code to generate the decryption key for it).
The on-demand scheme encrypts and decrypts code at the function granularity level. A function is decrypted only when it is called. A decryption key is generated on the fly (i.e., it is based on using properties of the functions that are already decrypted), and used to decrypt the function. When the function terminates its execution, it is encrypted again.

A hybrid technique combines bulk decryption and on-demand decryption, using the former for functions that are called by the program often, and the latter for the remaining functions. The hybrid technique provides a better performance than a pure on-demand decryption technique. On the other hand, the level of confidentiality protection that the hybrid technique offers is between that of the bulk decryption technique and on-demand decryption techniques.

The drawback of the solution is that the three techniques do not provide full protection against dynamic attacks. In all these techniques, there is a period when at least a part of an agent’s code is in a visited host’s memory in a decrypted form. During this period, this code can be attacked (e.g., to extract data or plaintext code, or to change code behavior).

5.6. Qualitative Evaluation of the Nine Solutions for Mobile Agents Providing Output to Visited Malicious Hosts

In this section we discuss the requirements for secure mobile agents solutions, evaluate qualitatively the nine solutions introduced above, and discuss the results of the evaluation.
5.6.1. Requirements for Secure Mobile Agent Solutions

We believe that a solution for protecting confidentiality of a mobile agent visiting malicious hosts (whether the visited hosts receive output from a mobile agent or not) must satisfy a set of requirements that we categorize into two classes: (i) requirements related to confidentiality-related mobile agent properties; and (ii) requirements related to the practicality of using a mobile agent solution (e.g., how easy it is to use it).

Recall our definitions of confidentiality-related properties for mobile agents:

1) Computation autonomy: A host visited by a mobile agent computes the entire code for the agent by itself. The visited host may request and use data (e.g., input, decryption keys) from other parties while computing the agent’s code.

2) Data autonomy: A host visited by a mobile agent computing code for the agent can use only data that it possesses (e.g., input, decryption keys). The visited host must not receive any data from any other party while computing the agent’s code.

Computation autonomy and data autonomy requirements are related to these confidentiality-related properties of mobile agents.

The requirements related to the practicality of using a mobile agent solution are:

1) Mobility: Mobility of a mobile agent is unrestricted. It is free to perform computations on arbitrary hosts, to deliver data to arbitrary hosts, or pass through arbitrary hosts on its way to a destination host.

---

40 Only confidentiality-related requirements—but not practicality requirements—are related to the properties of mobile agents listed in Subsection 5.1.4.
Table 5.3. Qualitative evaluation of existing solution against requirements

<table>
<thead>
<tr>
<th>Solution</th>
<th>Confidentiality-related Mobile Agent Requirements</th>
<th>Other Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computation Autonomy</td>
<td>Data Autonomy</td>
</tr>
<tr>
<td>S1: Protecting mobile agents using E-E-D property</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>S2: Protecting mobile agents using trusted hardware</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>S3: Protecting mobile agents using obfuscation</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>S4: Protecting mobile agents using coding theory and hardware-based methods</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>S5: Protecting mobile agents using garbled circuits and TTP</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>S6: Protecting mobile agents using garbled circuits and threshold-based multiparty computing</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>S7: Protecting mobile agents using privacy homomorphisms and function composition</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>S8: Protecting clueless mobile agents using environmental conditions</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>S9: Protecting mobile agents using polymorphism</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

2) *Ease of implementation:* A solution easy to implement is more likely to be developed and used.

3) *Providing output:* Mobile agents provide plaintext output to the visited hosts.

4) *Physical size of mobile agents:* The physical size of mobile agents should not make their transfer time among hosts unacceptably long. Physical agent size affects
scalability of a solution because it cannot support a significant number of large mobile agents.

5) **Avoiding special hardware:** Lower costs of purely software solutions will make their adoption faster than adoption of solutions with hardware-based elements (e.g., using coprocessors).

### 5.6.2. Qualitative Evaluation of Nine Solutions

In this subsection we justify our evaluation of the solutions shown in Table 5.3.

#### 5.6.2.1. Ratings for the Computation Autonomy Requirement

Solution S6 is rated “no” because computation of an agent’s function \( f \) (code) involves a set of \( t \) hosts. In more detail, computation of \( f \) is split between a set of mobile agents, and each agent is sent to a different host. The originator receives the partial outputs from all \( t \) hosts, combines the outputs, then decrypts the result to obtain the final plaintext output.

All the other solutions are rated “yes” because for each of them the visited host by itself computes function \( f \) for the visiting mobile agent.

#### 5.6.2.2. Ratings for the Data Autonomy Requirement

In Solutions S1, S2, S3, S4, and S7 data for computing a mobile agent’s code are obtained only locally, from the host visited by the agent. In particular: (a) the decryption keys required for Solutions S1, S2, and S4 are available at the visited hosts themselves, so data autonomy is not compromised; (b) Solution S3 does not need decryption keys.
Obfuscation approaches that use encryption/decryption include decryption keys inside their programs; and (c) Solution S7 does not need decryption keys since visited hosts do not receive plaintext output\textsuperscript{41}. (It returns an encrypted output to the originator. The originator uses it decryption key to decrypt the output of the mobile agent).

Solutions S5, S6, S8, S9 are rated “no” because each of them requires that a mobile agent requests decryption keys from one or more entities (hosts, a user) other than the visited host. To be more specific: (a) S5 requests the decryption keys associated with its input from a TTP; (b) S6 requests the decryption keys associated with its inputs from \(t\) hosts, where \(t\) is a threshold value for the solution; (c) S8 requires an environment timing condition to occur which is provided by an entity other than the visited host; and (d) S9 requests decryption keys from a user or from the operating system.\textsuperscript{42}

5.6.2.3. Ratings for the Mobility Requirement

We rate all described solutions “yes” because we are convinced that they allow for unrestricted agent mobility. (In general, there are mobile-agent-based solutions that use stationary not mobile agents, such agents that behave like a standalone application or a service).

\textsuperscript{41} Note that we included S7 in this evaluation for completeness of the evaluation since the solution is very popular for protecting mobile agent data.

\textsuperscript{42} Solution S9 can use one of two strategies for fetching description keys. In the first strategy the keys are fetched from a user or the operating system where the solution is executed. In the second strategy the decryption key is embedded into the code [CEJO2002b]. Using the second strategy, S9 satisfies the data autonomy. However in this evaluation we consider the general case which is the first strategy. Therefore, the solution does not satisfy the criterion.
5.6.2.4. Ratings for the Ease of Implementation Requirement

Solution S3 is rated “no” because obfuscating mobile agents is difficult (cf. Subsection 5.3.3). Solution S7 is rated “no” because we did not find any secure privacy homomorphisms that support both algebraic and Boolean operations (cf. Subsection 5.3.7).

We see no significant implementation challenges for the remaining solutions, so we rate them “yes.”

5.6.2.5. Ratings for the Providing Output Requirement

Solution S7 is rated “no” because in this solution a visited host is not given plaintext output (only the originator can decrypt the output).

In all other solutions, the visited host receives plaintext output from the visiting mobile agent, so they are rated “yes.”

5.6.2.6. Ratings for the Physical Size of Mobile Agents Requirement

Solution S1 is rated “no” because it requires having a copy of a mobile agent for each host that the mobile agent might visit. Each per-host copy is encrypted with the encryption key of the corresponding host. All copies are bundled together, and each host receives the whole bundle of copies.

The remaining solutions avoid having multiple copies of a mobile agent for multiple (or all) hosts, so they are rated “yes.”
5.6.2.7. Ratings for the Avoiding Special Hardware Requirement

Solutions S2 and S4 are rated “no” because they require that visited hosts possess a special hardware, such as a co-processor.

We rate the rest of the solutions “yes,” since they do not require special hardware.

5.6.3. Summary of Qualitative Evaluation of Nine Solutions

Table 5.3 summarizes our qualitative evaluation of the reviewed Solutions S1 through S9. The evaluation uses the requirements from the preceding subsection.

As shown in Table 5.3, only five solutions— S1, S2, S3, S4, and S7—satisfy both confidentiality-related mobile agent properties. Unfortunately, each of these solutions has serious deficiencies. Solution S1—i.e., protecting mobile agents using the E-E-D property—has two serious drawbacks. First, it is the only one that does not protect agents’ code (as we know from Section 5.5.1). Second, it has a scalability problem since it is based on assigning a key to each user who wants to use a mobile agent. If copies for \( n \) users are included in the same mobile agent, the agent must be encrypted \( n \) times.

Solution S2—i.e., protecting mobile agents using hardware-based secure computing—requires that only hosts equipped with trusted hardware can execute mobile agents.

Solution S3—i.e., protecting mobile agents using obfuscation—is not available yet due to the nonexistence of a practical secure program obfuscation solution (cf. Subsection 5.3.3).

Solution S4—i.e., protecting mobile agents using coding theory and hardware-based methods—requires that only hosts that are equipped with trusted hardware can execute mobile agents (similar as S2).
Solution S7—i.e., protecting mobile agents using privacy homomorphisms and function composition—does not provide output to visited hosts.

5.7. Conclusions

A mobile agent is a software object able to perform computations on visited hosts, transfer itself or be transferred by its host to another host, and interact with visited hosts taking advantage of their capabilities.

Mobile agents and active bundles have similar a structure since both include data and code. The goal is to investigate solutions for protecting confidentiality of mobile agents in cases when they provide output to visited malicious hosts. The intuition is that if a solution exists and it satisfies (i) the two critical mobile agent confidentiality-related properties (i.e., computation autonomy and data autonomy), and (ii) practicality requirements, then this solution could be used for active bundles.

In this work we present and evaluate qualitatively a set of nine solutions known in the literature for protecting confidentiality of mobile agents that reveal output to visited hosts. As mentioned, each solution uses at least one secure computing method. The evaluated solutions are: (S1) protecting mobile agents using the E-E-D property, (S2) protecting mobile agents using hardware-based secure computing, (S3) protecting mobile agents using obfuscation, (S4) protecting mobile agents using coding theory and hardware-based methods, (S5) protecting mobile agents using garbled circuits and TTPs, (S6) protecting mobile agents using garbled circuits and threshold-based multi-party computing, (S7) protecting mobile agents using privacy homomorphisms and function composition, (S8)
protecting clueless mobile agents with environmental conditions, and (S9) protecting mobile agents using polymorphism.

The main contribution of this chapter is answering the two critical questions: (1) Are there any available solutions for protecting confidentiality of code and carried data of mobile agents providing output to visited hosts that satisfy the two critical confidentiality-related mobile agent properties: computation autonomy and data autonomy? (2) Are any of these solutions practical?

The answer to the first question is “Yes,” but the response to the second question is “No.” In more detail, there are five solutions that satisfy the two critical mobile agent confidentiality-related properties but these solutions do not satisfy the practicality requirements for the following reasons. The Solution S1 does not protect agents’ code and is not scalable, Solutions S2 and S4 do not apply to an arbitrary host (require trusted hardware), Solution S3 is based on a method that is not practical yet, and Solution S7 does not provide plaintext output to visited hosts.

After evaluating solutions S1 – S9, our answer is that—to the best of our knowledge—there is no known practical solution that: (i) protects confidentiality of code and carried data of mobile agents providing output to visited hosts, and (ii) satisfies the two critical confidentiality-related mobile agent properties: computation autonomy and data autonomy.
CHAPTER 6

ABTTP: A TTP-BASED PROTOTYPE
IMPLEMENTING THE ACTIVE BUNDLE SCHEME

6.1. Introduction

This chapter describes the prototype, named ABTTP, for the active bundle scheme using a trusted third party (TTP). The active bundle scheme protects sensitive data from unauthorized disclosures of sensitive data. The TTP’s main role is to store decryption keys and provide them to entities authorized to access data protected by active bundles.

ABTTP achieves its data protection goals by: (i) enforcing privacy policies defined as part of active bundle’s metadata; (ii) activating protection mechanisms when the bundle’s data, and/or metadata is tampered with; and (iii) recording activities on an active bundle (e.g., its transfers between hosts) for the bundle’s audits. The scheme provides protection from hosts that might attempt to violate privacy policies associated with sensitive data. Mechanisms used to this end include evaporation and apoptosis, both described in Chapter 4.

ABTTP demonstrates how the active bundle scheme can be used, and shows its practicality. It assumes the existence of a TTP and an Internet connection between the TTP and hosts using active bundles. As the prototype shows, when this assumption is satisfied, the scheme protects confidentiality of sensitive data from unauthorized entities.
This chapter is organized as follows. Section 6.2 portrays the threat model for TTP-based implementations of the active bundle scheme. Section 6.3 places the ABTTP prototype within the mobile agent architecture framework. Section 6.4 details the deployment setup for ABTTP, and Section 6.5 describes the configuration setup. Section 6.6 describes the experiments run by us on ABTTP. Section 6.7 discusses the experiments. Section 6.8 discusses future extension to the prototype. Section 6.9 concludes the chapter.

6.2. Threat Model for TTP-based Implementations of the Active Bundle Scheme

Sensitive data can be compromised in different ways. ABTTP protects them only from the following set of confidentiality threats: unauthorized data disclosures, unauthorized data modifications, unauthorized data disseminations. Other threats, which are security threats but not confidentiality threats, are beyond the scope of this research. Examples of such threats include: data inferences, Sybil attacks, and masquerade attacks.

ABTTP ensures enforcement of confidentiality policies associated with sensitive data even on untrusted hosts. This is accomplished since untrusted hosts cannot obtain decryption keys for protected data from the TTP.

The solution relies on the following assumptions (accompanied here with brief explanations or discussion):

A1) Access to a TTP that stores decryption keys and provides them to enabled active bundles is assured.

A2) The TTP can obtain from a trust server information indicating trustworthiness of any host visited by an active bundle.
Host trustworthiness information can be obtained through different approaches, including remote attestation of hosts [CGLH10]. It is further assumed that the trust level of a visited host is indicated with a single numerical value.

A3) Any host receiving an active bundle correctly executes the code included in the VM of the bundle; in particular, it does not disable execution of this code.

We require that a trust evaluation server (TES) notifies the security server that a host is not trusted when TES evaluates that the host does not execute the bundle’s code correctly. If the host stopped execution of the code before apoptosis is activated, this information cannot be used to activate apoptosis (since the code that could activate apoptosis is already stopped). However, after receiving such a notification, the security server denies the host’s requests for decryption keys, which prevents the host from accessing the bundle’s data.

A host that tampers with execution may disallow posing requests for decryption keys from the security server by the bundle’s code. In such case it cannot access the sensitive data included in the active bundle.

A4) There is a secure communication channel between active bundles and TTPs.

We see the following limitations of the current ABTTP implementation:

1) Trustworthiness of a host is dynamic; that is, it may change over time. The most dangerous situation occurs when a host authorized by a bundle’s privacy policy to access some (or even all) of its data becomes a host that is not authorized to access these data.

If the host obtained decryption keys when it was still authorized, it can keep accessing the bundle’s data after losing its authorization. (Revoking decryption keys
fast enough is a possible solution.) If the host obtained data when it was still authorized, it can keep disseminating these data after it lost its authorization. (We see no solution for this situation: the genie is already out of the bottle). In both situations, the host is analogous to a person that breaks one’s trust.

2) Security server and hosts receiving active bundles authenticate active bundles—possibly by using certificates for active bundles. The certificates are issued by other TTPs.

3) A malicious host (i.e., attacker) can copy an active bundle before activating it. As long as the host does not obtain decryption keys from a TTP, it cannot access data stored in the ciphertext form within the bundle.

4) ABTTP does not protect from side-channel attacks. For example, a host may cheat by accessing the memory used by the bundle. It can get decryption keys, and use them to decrypt data that the associated policies do not authorize it to access.

6.3. ABTTP in the Mobile Agent Architecture Framework

This section presents in the first subsection the tools and framework used in the prototype, in the second subsection—the architecture of the ABTTP prototype, and in the third subsection—the behavior of the AB components.
6.3.1. Tools and Framework used in the Prototype

We have developed ABTTP in the context of the mobile agent framework\(^{43}\) known as JADE [BCG07]. Other major tools used for implementing the prototype include the programming language Java 1.6.01 [Java10], the development environment Eclipse 3.5.2 [Ecli10], and Java cryptographic libraries [Knud98]. JADE and Java cryptographic libraries are discussed next.

6.3.1.1. The Mobile Agent Framework JADE

JADE is a software framework that provides basic mobile agent middleware functionalities independent of the specific application. These functionalities simplify realizations of distributed applications that are based on the mobile agent paradigm [BCG07]. Fig. 6.1 illustrates the architectural elements of JADE. The *JADE framework* (i.e., libraries) enables the development of multi-agents applications. The *JADE platform* enables deployment of multi-agents applications on one or more hosts.

JADE is developed using the object-oriented language Java. The JADE platform (a software platform) is composed of *agent containers* (or just *containers*) that can be distributed over network nodes. A *platform* is a distributed application that includes containers joining and leaving it dynamically. A *container* is a Java process, an object instantiated by a Java program, that provides the JADE run-time services and all other services needed for hosting and executing agents. Containers can be distributed on a

---

\(^{43}\) A *mobile agent* (a.k.a. a *mobile object*) is a software object able to perform computations on visited hosts, transport itself from one host to another, and interact with and use capabilities of visited hosts. More details about mobile agent were provided in Chapter 5.
JADE platform among a set of hosts that communicate with each others. The *main container* is a special container that represents the bootstrap point of a JADE platform (i.e., a distributed application). It is the first container to be launched by a platform, and all other containers must register with the main container [BCG07].

The main container provides a number of functionalities, including the following ones [BCG07]:

1) *Hosting the Directory Facilitator (DF).* DF is a special agent that provides default yellow-page services for the platform. The yellow pages service is used by agents searching for available service, or wishing to register their own services.

2) *Hosting the Agent Management System (AMS).* AMS is a special agent that supervises the entire platform; among others, it provides management of agents’ lifecycles, and
white-page services. AMS is the contact point for all agents that need to interact with each other.

3) **Managing the Local Agent Descriptor Table (LADT).** LADT is a registry of agents available in the local container. Management of LADT includes maintaining their current status.

4) **Managing the Global Agent Descriptor Table (GADT).** GADT is a registry of agents for a platform. Management of GADT includes maintaining their current status and locations. Each container has its own LADT, while there is only one GADT, which includes information from all LADTs. The GADT cache is a copy of GADT at a specific time (GADT is synchronized periodically with all LADT).

5) **Managing the Container Table (CT).** CT is a registry of containers. Management of CT includes maintaining references and transport addresses for all containers for a given platform.

A mobile agent in JADE is a single-threaded Java program that has a set of behaviors. A behavior is a Java object that runs until it is finished. A behavior object executes a single task, such as “Send this message,” or a complex task in a thread. When the behavior object finishes its execution, it gives the control back to its agent [Este10].

A mobile agent communicates with other agents using asynchronous message passing (no waiting for the reply from a called function on another agent). Each agent has a mailbox where messages sent by other agents are posted, until they are picked up for processing by the agent. Messages are structured using ontologies for agent communications. (An ontology provides a description of concepts and relationships between them.) Mobile agents communicating with each other must know this ontology;
otherwise they would be unable to encode and decode the messages they exchange [Este10].

6.3.1.2. *Java Cryptographic Libraries*

Java cryptographic libraries are composed of two components [KNUD98]:

1) *Java Cryptography Architecture (JCA)*: JCA is a library included in the Java Development Kit (JDK). It *defines* cryptographic concepts; the descriptions (but not implementations) are encapsulated in classes that compose `java.security` and `javax.crypto` packages.

2) *Java Cryptography Extension (JCE)*: JCE is an extension library, including algorithms absent from JCA (and, thus, from JDK). JCE implements the descriptions defined in JCA in the namespace `javax.crypto`. Export restrictions allow for using JCE only in the United States and Canada; other implementations of JCE must be used elsewhere.

The libraries are used in the ABTTP prototype for: (i) encrypting and decrypting active bundles, (ii) generating encryption and signature keys, (iii) computing the cryptographic hash values for sensitive data included in active bundles, and (iv) signing active bundles.

6.3.2. *Description of the ABTTP Architecture*

For our and readers’ convenience, we often use the “AB” acronym to denote “Active Bundles.”

The components of our prototype, illustrated in Figure 6.2, are distributed among four containers:
1) Main Container—AB Coordinator includes Directory Facilitator (DF).

2) Container 1—Client Application includes AB Creator that creates active bundles; one active bundle is shown explicitly in the figure. (Note that Active Bundle or AB is a mobile agent, not a Java class).

3) Container 2—AB Destination (ABD) includes disseminated active bundles. One active bundle that arrives at ABD is shown explicitly in the figure.

4) Container 3—AB Services includes Security Services Agent (SSA), Trust Evaluation Agent (TEA), and Audit Services Agents (ASA). All three are TTPs.

Figure 6.2 shows (with the arrow labeled “Move”) an active bundle being disseminated from its creator in Container 1—Client Application to Container 2—AB Destination (ABD). In ABTTP, the destination is specified when an active bundle is created by AB Creator.

Figure 6.2. UML component diagram [BSSV10] of the ABTTP prototype: a TTP-based active bundle implementation
6.3.2.1. **Main Container—AB Coordinator**

Main Container—AB Coordinator hosts DF, which provides a yellow-page\(^{44}\) service. The service is used by mobile agents to register and deregister their services, to modify their registered service descriptions, and to search for needed services.

In this prototype, DF registers four agents: the single active bundle AB, and three AB services: SSA, TEA and ASA. The agents communicate in such a way that messages exchanged between them are delivered to a destination agent even when it moves around, visiting different hosts. A non registered agent cannot receive messages addressed to it because its location (its host and its container) cannot be found.

6.3.2.2. **Container 1—Client Application**

*Container 1—Client Application* hosts AB Creator. AB Creator accepts a user’s input for an active bundle—including sensitive data, metadata, and the indication of the destination for the bundle. It creates an active bundle, filling its structure with the contents provided by a user (who becomes the owner of this bundle). AB Creator also includes a set of functions that compose the bundle’s VM. Next, it registers the complete bundle with DF.

In the prototype we use only a subset of the elements of an active bundle (as described in Subsection 4.1). This is consistent with the role of the prototype: it is built to prove feasibility of the active bundle scheme.

\(^{44}\) A yellow page is a registry of entries which associate service descriptions to agent IDs [BCG2007].
6.3.2.3. Container 2—Active Bundle Destination

The only function of Container 2—AB Destination is receiving active bundles. We can create an arbitrary number of AB Destination instances. Each AB Destination instance must be enclosed in a container that has a unique name (e.g., the first instance is enclosed in Container 22, the second instance—in Container 25, and the third instance—in Container 37). Any active bundle can move to any AB Destination (as decided by the AB owner).

6.3.2.4. Container 3—Active Bundle Services

Container 3—Active Bundle Services includes three service agents: SSA, TEA, and ASA. The first agent, SSA (Security Services Agent), maintains a database of security information on active bundles. This information is used for encrypting and decrypting contents of active bundles. SSA keeps the following information on each active bundle: name of the bundle, its decryption key, the trust level that a host must satisfy to use the active bundle. Note that this information is different from the bundle’s metadata. Metadata is contained within the active bundle itself while this security information is stored by SSA (on the SSA’s host).

SSA stores identity-related data for active bundles in a file as a hash table object. We chose the hash table data structure since it speeds accesses to SSA identity-related data. The hash table uses the active bundle name as an entry key.

The second agent, TEA (Trust Evaluation Agent), answers requests from SSA about the current trust level for a specified host. Since our goal is proving feasibility of the active bundle scheme, not developing trust management or trust evaluation solutions
(which is a research subject in itself), we do not implement a full-fledged TEA. Instead, we simulate a TEA service by assigning a random number as a response to a request for a host’s trust level. In actual active bundle scheme implementations, building TEA will require implementation of trust management based on trust measurement and estimation [Zhon05].

The third agent, ASA (Audit Services Agent), monitors activities of active bundles. It receives audit information from active bundles, and records this information into a file for possible analysis by authorized entities⁴⁵ (e.g., owners, auditors, or forensic analysis agents). In this prototype, ASA records every arrival event—when an active bundle (identified by its name) arrives at a destination host. The audit service in the prototype can be easily extended to facilitate auditing other activities of active bundles, such as access to their data by recipients.

SSA, TEA, and ASA share configuration management data. These data are currently included in a Java class; values for a set of system parameters are specified there (discussed below in Subsection 6.5.1), and can be accessed by methods of the class.

SSA, TEA, and ASA also share the same logging class. The logging tool in ABTTP is a Java class that provides a method to log important events during execution of the prototype for debugging purposes. Log information is output to both a screen window and a log file.

6.3.2.5. Active Bundles

ABTTP implements an active bundle as a mobile agent with a set of attributes and operations. Note that a mobile agent includes functions that support operations of active

⁴⁵ As a future extension of this work, we plan to realize a set of tools to support audits.
bundles. An active bundle is an instance of this mobile agent Java class. Therefore, the bundle includes these functions as the bundle’s VM.

6.3.3. Behavior of the AB Components

Behavior of an active bundle can be summarized as a sequence of three steps:

1) AB Creator creates an active bundle AB. This includes providing the owner with a list of possible destinations for the bundle, and requesting the owner for selecting one.

2) Upon receiving the destination choice, the active bundle AB triggers constructing of an algorithm for itself. Construction steps for the AB includes encrypting and signing data included in it (details are given in Subsection 6.3.3.3). Then, after receiving its destination form its owner, the AB moves\(^\text{46}\) to the destination host. Note that an active bundle can construct itself before its destination is given to it.

3) After arriving at the destination host, the AB triggers its own enabling algorithm (described in Subsection 6.3.3.3).

In the following we describe in more details the algorithms for creating, constructing and enabling an active bundle.

6.3.3.1. Creating an Active Bundle

AB Creator creates an active bundle “object” of the Java class “active bundle.” This class includes code that becomes the virtual machine (VM) of the active bundle when the

\(^{46}\) ABTTP uses the push dissemination style (cf. Section 1.1).
class is instantiated.

The creation process involves an owner of sensitive data providing AB Creator with sensitive data and metadata (including a privacy policy). AB Creator creates an active bundle by putting together data, metadata, and adding a VM to the bundle. The bundle becomes now a (potentially) active entity that can perform its activities using its own VM.

We assure that a host receiving an active bundle cannot change execution of the bundle’s VM since it is executed by hosts that have TPM.

6.3.3.2. Constructing an Active Bundle

The steps of the algorithm for constructing active bundles, summarized in Fig. 6.3, are:

1) An active bundle requests by itself for and gets from SSA two pairs of public/private keys. The first pair of keys is used for encrypting an active bundle; and the second pair of keys is used for signing data included in the bundle, or verifying such data signature. The reason for having two key pairs is to prevent attackers from modifying the bundle’s sensitive data, and signing it again with the public key of the data owner.

2) The active bundle sends a request to SSA (which is a TTP), asking it to store the bundle’s identity-related data. These data include the name of the bundle, its decryption key, and the minimum required trust level for individual hosts (each host’s trust level requirement must be satisfied before it may access the bundle’s data). The goal of using SSA is to keep the decryption keys and other identity-related data for active bundles in a trusted location. The decryption keys are issued by SSA only to hosts that are eligible to access the bundle, which in this simple
A prototype implementation means hosts with their trust level exceeding the minimum required by the bundle.

3) The active bundle computes a hash value for sensitive data, and signs them (on behalf of its owner) using signature keys. Signature keys are not disclosed to containers receiving active bundles, and are available only at the client application used by the owner. The signature certifies that data are from its owner.

4) The active bundle encrypts sensitive data using the encryption key.

![Diagram](image-url)

**Figure 6.3. UML activity diagram for constructing an active bundle**

6.3.3.3. *Enabling an Active Bundle*

After arriving at its destination (*Container 2—Active Bundle Destination*), an active bundle *enables* itself. The enabling algorithm for an active bundle, illustrated in Fig. 6.4, can be summarized as follows (with steps referring to numerical labels in the figure).

In Step 1, an active bundle sends a request to SSA asking for its identity-related information. (Recall that identity-related information for a bundle includes the decryption key, the signature verification key, and the required trust level.)
The request includes the identity of the bundle’s current container. (The bundle knows
the identity of its container because it was specified as its destination when the bundle’s
move was ordered by the owner.) SSA replies with all identity-related information
(including the decryption key) if the destination host has a satisfying trust level (higher
than the trust level required by the bundle to access its data). Otherwise, it provides only
the host’s trust level. It does not provide the decryption key.

Figure 6.4. UML activity diagram for enabling an active bundle

In Step 2, an active bundle checks if the hosts’ trust level is sufficient for accessing the
bundle’s data. If not, the bundle apoptosizes (Step 3); otherwise, it executes Step 4.
In Step 4, an active bundle checks the integrity of its sensitive data by computing the hash value for these data. Then, it verifies its signed hash value by comparing it to the computed hash value. If the verification fails, the bundle apoptosizes (Step 5); otherwise, the bundle uses the decryption key received in Step 1 to decrypt its sensitive data (Step 6).

In Step 7, the active bundle enforces its privacy policies. In this prototype, we do not actually develop and use privacy policies. (It is not necessary in this proof-of-feasibility prototype. A future extension of ABTTP, making it more comprehensive and realistic, will include components that assist in developing privacy policies.)

In Step 8, the active bundle provides the output to the visited host.

In Step 9, the active bundle sends audit information to ASA. This information includes the bundle’s name, its host’s identity, and the name of the audited event that is being reported. The only event that we trace in the current version of ABTTP is “arrive at a host.”

### 6.4. Deployment Setup

**By deployment setup we mean distribution of components among hosts and sessions.** The login process creates an initial process and a session. A *session* is a collection of processes that were started directly or indirectly by the initial process.

In this setup, we grouped the components into four groups. Fig. 6.5 shows the deployment diagram of ABTTP using the deployment setup. In this deployment, each group of components is located in a different container but all components are executed by the same computer, which is the server. The first group, Client Application, is composed of AB Creator, and is located in Container 1 (Unix Server-Session 1). The second group,
AB Destination, includes the component AB Destination, and is located in Container 2 (Unix Server-Session 2). The third group, AB Coordinator, includes only DF (which is a JADE server), and is located in Main Container (Unix Server-Session 3). The fourth group, AB Services, is composed of SSA, TEA, and ASA; it is located in Container 3 (Unix Server-Session 4).

In an experiment with this deployment setup, AB Creator creates active bundles within Container 1. Then the bundles move to Container 2 where they enable themselves.

![Diagram of the deployment setup of ABTTP](image)

**Figure 6.5. Diagram of the deployment setup of ABTTP**

6.5. **Configuration Setup**

A *configuration setup* determines the values of parameters (a.k.a. configuration variables) used by ABTTP, such as network port numbers, and names of files used by the programs.
The ABTTP prototype is composed of a set of components distributed among hosts, with parameters of the prototype specified by the configuration setup.

The configuration variables are grouped into three groups. Each group of variables is used by one group of components. In the following we describe the three configuration groups used by ABTTP.

Table 6.1 describes the configuration variables for the ABC component. The table describes the four parameters that the components use which we describe in the following. A value for parameter “LogFile” specifies the name and path of the logging file. A logging file is used by the components to store data that describe the log of activities on an active bundle (e.g., encrypting, decrypting, signing, moving, etc.).

Values for parameters “host” and “port” specify the location (IP address), and the communication port of the Active Bundle Coordinator respectively. A value for parameter “Fname” specifies the name of the Jade Platform that the components can communicate with. Note that this parameter is not used explicitly by our ABTTP components. It is used by JADE for coordinating the communication between mobile agents.

### Table 6.1. Configuration variables for the ABC component

<table>
<thead>
<tr>
<th>Configuration variable</th>
<th>Description</th>
<th>Example of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogFile</td>
<td>Name and path of the file where the application logs the events.</td>
<td>ABCCreatorLog.txt</td>
</tr>
<tr>
<td>host</td>
<td>The IP of the host of the Active Bundle Coordinator.</td>
<td>141.001.143.001</td>
</tr>
<tr>
<td>Fname</td>
<td>Name of the platform that the client should connect to.</td>
<td>ABFramework</td>
</tr>
<tr>
<td>port</td>
<td>The port that the client should connect to in order to communicate with the Active Bundle Coordinator.</td>
<td>1099</td>
</tr>
</tbody>
</table>
Table 6.2. Configuration variables for the AB services component

<table>
<thead>
<tr>
<th>Configuration variable</th>
<th>Description</th>
<th>Example of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABIdentitiesStore</td>
<td>Name and path of the file where the information about the identities of created active bundle including keys are stored.</td>
<td>ABkeys.obj</td>
</tr>
<tr>
<td>LogFile</td>
<td>Name and path of the file where the application logs events.</td>
<td>CoordinatorLog.txt</td>
</tr>
<tr>
<td>AuditFileName</td>
<td>Name and path of the file where the audit information are stored.</td>
<td>AuditLog.txt</td>
</tr>
<tr>
<td>host</td>
<td>The IP of the host of the Active Bundle Coordinator.</td>
<td>141.001.143.001</td>
</tr>
<tr>
<td>Fname</td>
<td>Name of the platform that the client should connect to.</td>
<td>ABFramework</td>
</tr>
<tr>
<td>port</td>
<td>The port that the client should connect to in order to communicate with the Active Bundle Coordinator.</td>
<td>1099</td>
</tr>
</tbody>
</table>

Table 6.2 describes the configuration variables for the Active Bundle Services component. The table describes the six parameters that the components use which we describe in the following. A value for parameter “ABIdentitiesStore” specifies a name and path of the file used by SSA to store and retrieve identity-related data about active bundles.

A value for parameter “LogFile” specifies the name and path of the logging file. A logging file is used by the components to store data that describe a log of activities of the services (e.g., receiving messages, sending messages).

A value for parameter “AuditFileName” specifies the name and path of the audit file. An audit file is used by the component to store data that describe audit information about important events on active bundle. In ABTTP we record currently only active bundles’ move to ABDs.

Values for parameters “host” and “port” specify the location (IP address), and the communication port of the Active Bundle Coordinator. A value for parameter “Fname” specifies the name of the Jade Platform that the components can communicate with.
The difference between a log file content and an audit file content is in the type of data that each file stores. Data in a log file could be used to trace a sequence of events (e.g., for debugging). This data is for only a period of time; it is a temporal data since it is deleted after a time period. The log file could be used by a developer or administrator for debugging.

In contrast, audit file content are data that could be archived and saved for a long time. The purpose of the audit file is to have a trace of activities that could be used for forensic investigations, and/or for record management.

Table 6.3 describes the configuration variables for the Client Application. The table describes the four parameter variables that the component uses. The variables have the same description as the ones described in Table 6.1.

<table>
<thead>
<tr>
<th>Configuration variable</th>
<th>Description</th>
<th>Example of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogFile</td>
<td>Name and path of the file where the application will log events</td>
<td>ABHostLog.txt</td>
</tr>
<tr>
<td>host</td>
<td>The IP of the host of the Active Bundle Coordinator</td>
<td>141.001.143.001</td>
</tr>
<tr>
<td>Fname</td>
<td>Name of the platform that the client should connect to</td>
<td>ABFramework</td>
</tr>
<tr>
<td>port</td>
<td>The port that the client should connect to in order to communicate with the Active Bundle Coordinator</td>
<td>1099</td>
</tr>
</tbody>
</table>

6.6. Description of the Experiments

This section describes two experiments that we performed on ABTTP.
a) Jade Server
Started with: java jade.Boot -name ABFramework

b) Active Bundle services
Started with: java CreateServers
c) Creation of an active bundle

Started with: java ActiveBundleCreator

Figure 6.6. Sample screen shots for the executing prototype

Fig. 6.6 shows simple screen shots for executing the prototype. The first session (Session 1 in Fig. 6.5) executes the AB Creation component. The command to execute the component is “java ActiveBundleCreator.” The second session (Session 2 in Fig. 6.5) executes the AB Destination component. The command to execute the component is “java ActiveBundleDestination.” The third session (Session 3 in Fig. 6.5) executes JADE Server, which runs the AB Coordinator. The command to execute the component is “java jade.Boot –name ABFramework.” The fourth session (Session 4 in Fig. 6.5) executes AB Services. The command to execute the component is “java CreateServers.”

The component ABC prompts the owner for the name of the active bundle, the sensitive data, the metadata, the required trust level and the role\textsuperscript{47} of the destination hosts. The component initializes and constructs an active bundle and gives it the input of the user. Subsection 6.3.3.1, and Subsection 6.3.3.2 describe the algorithms for initializing and constructing, respectively, active bundles.

\textsuperscript{47} Note AB Creator requests from the owner the role of the AB Destination (in the sense of role based access control) but we do not use this information further.
Figure 6.7 shows an example of an active bundle getting the destination host from the owner. It shows the exchange of information between an active bundle and the owner before the move of the active bundle to a destination container. The steps are:

1) The active bundle lists the index and location of available containers where the bundle could move to.

2) The bundle prompts the owner to enter the index of the location that he wants the active bundle to move to. (The index is assigned to each destination to simplify the prototype.)

3) The owner enters his destination choice.

4) The active bundle displays a message confirming that a move to a specified destination is ordered. This message does not confirm that the bundle reached the destination. It only indicates that the bundle is ready to move to the destination.
a) Log for AB Services

b) Log for AB Destination

Figure 6.8. Sample screen shots for the executing prototype for Experiment 1
Upon receiving the index of the destination container, the active bundle constructs itself. Then, it moves to the destination container. After arriving into the destination container, the active bundle enables itself.

To keep the prototype simple, TEA generates trust level values between 0 and 5. These values are distributed based on the normal distribution [BCNN05].

When an active bundle enables itself, it may decrypt itself (as presented in Experiment 1) or apoptosize (as presented in Experiment 2). In the following we describe both experiments.

### 6.6.1. Experiment 1: Decryption of an Active Bundle

In this experiment we select value “1” as the required trust level to access the active bundle. TEA generates value “3” as the trust level of the host of the AB Destination. Therefore, the answer to the question “Does host’s trust level exceed the required minimum?” is “Yes” (see Fig. 6.4). So, the host is allowed to access some or all sensitive data included in the bundle. Then, the VM of the bundle decrypts its sensitive data (and the visited host can see them).

Fig. 6.8.a displays the log of AB Services, including a request for a host’s trust level and SSA’s response. Fig. 6.8.b displays the log of AB Destination, including the records of a successful decryption and an integrity check of the bundle.
Active bundles services. The value of the required trust level is 5, and the value of the host’s trust level is 4.

Active bundle. Note that the active bundle in this case commits apoptosis.

Figure 6.9. Sample screen shots for the executing prototype for Experiment 2

6.6.2. **Experiment 2: Apoptosis of an Active Bundle**

In this experiment, the required trust level to access the active bundle is set to value 5. TEA generates value 4 as the current trust level of the host with the destination container.
Therefore, the answer to the question “Does host’s trust level exceed the required minimum?” is “No” (see Fig. 6.4). So, the host is not allowed to access the bundle’s data. Furthermore, the threatened bundle apoptosizes.

Fig. 6.9.a displays the log of AB Services, including records of a request for a host’s trust level and the SSA’s response. Fig. 6.9.b displays the log of AB Destination, including the record of the bundle’s apoptosis.

6.7. Discussion of the Experiments

The active bundle approach provides three levels of confidentiality protection for bundle’s sensitive data throughout their lifecycle. Having three levels improves the protection significantly.

The first level of confidentiality protection relies on using a TTP. A TTP has two major roles:

1) Assure that a destination host has a minimum level of trust as required by the owner of an active bundle. The TTP supports SSA and TEA.

2) Provide decryption keys for accessing the bundle’s data only when the trust level of the destination host is sufficient to satisfy requirements imposed by the active bundle.

These roles assure enforcing an owner’s access control policies for disseminated active bundles throughout the bundles’ lifecycle, even when the bundles are far away from the owner. (The prototype can be improved so that the owner will be able to withdraw a decryption key and/or change the required trust level for active bundles by exchanging messages with the TTP.)
The second level of confidentiality protection relies on having different confidentiality protection levels for different portions of data within an active bundle. A decryption key received by an active bundle from the TTP can enable a visited host’s access to only a portion of a bundle’s sensitive data; privacy policies decide which portions of data can be accessed.

The third level of confidentiality protection relies on using bundle’s VM. The VM receives needed decryption keys from the TTP. Depending on the visited host trust level (insufficient or sufficient), the VM activates one of the following two actions: (1) apoptosis (for an insufficient host’s trust level), or (2) decryption of sensitive data and enforcement of the bundle’s privacy policy w.r.t. portions of data (for the sufficient host’s trust level).

6.8. Future ABTTP Extensions

ABTTP, being a proof-of-feasibility prototype for the active bundle scheme, does not implement many mechanisms that a fully-functional system would. ABTTP has many limitations that we plan to eliminate in our future work:

1) Recall that in the prototype, privacy policies defined for an active bundle are rudimentary: they only determine the required trust level for hosts wishing to access a bundle’s data. Also, the privacy policies are static: they do not change once a bundle is disseminated. However, the owner may want to change the policies of his active bundle. A future work will address both issues.
2) Future extensions of the AB Creator will add an integrity check mechanism for VMs, assuring that the VM code is the same as originally created by the AB Creator. This will prevent running a bundle’s VM if it were tampered with by any host.

3) Recall that ABTTP simulates the use of a trust evaluation server. Future extensions include developing a trust management mechanism for evaluating host trustworthiness based on existing approaches for trust management, such as remote attestation [CGLH10], Zhong’s trust framework [Zhon05], and a self-organizing peer-to-peer trust model [CaBh06].

6.9. Conclusions

The main challenges in information sharing area are limitations of mechanisms for protecting confidentiality of sensitive data throughout their lifecycle. An owner of sensitive data may not be able to identify all entities that are allowed to access these data (i.e., may not be able to enumerate all authorized data recipients). In such a case, the approach using keys cannot be used. The reason is that the unknown entities cannot access data since no decryption keys are prepared for them.

The alternative approach is to use privacy policies as the basis for protection of confidentiality of sensitive data. However, attaching privacy policies to data is not sufficient if an attacker can access data without enforcing their privacy policies. The active bundle scheme, presented in this Thesis, goes further, by attaching to the data also an enforcement mechanism, and trying to make both the data and the enforcement mechanism (VM) inseparable from the data.
This chapter describes ABTTP—our TTP-based prototype implementation of the active bundle scheme (where the TTP plays the role of a security server).

The TTP assures that a destination host for an active bundle is allowed to access its data only if it has a minimum level of trust as required by the owner of the bundle. ABTTP uses a comprehensive numerical value indicating the trust level of the visited host.

The ABTTP prototype demonstrates: (1) encrypting sensitive data and storing decryption keys at a TTP; (2) signing data to ensure their integrity; (3) activating apoptosis when a host receiving the bundle is not allowed to access any portion of the bundle’s data due to its sufficient trust level; (4) decrypting data, checking integrity of data and simulating enforcement of privacy policies when the receiving host is allowed to access a portion or all data; and (5) collecting audit information and storing it by Audit Service Agent (ASA) on the TTP (storing on the TTP minimizes risks of manipulation or forging of audit information by the owner; this assures correctness of the data should they be used by authorized parties, such as an IT auditor, a forensic investigation officer, or a court official).

The ABTTP prototype demonstrates the active bundle scheme and shows its practicality. It shows that the scheme protects confidentiality of sensitive data from accesses by unauthorized entities.
7.1. Introduction

We define an autonomous application as a set of programs communicating with each other in such a way that the application is able to self-govern and is independent in its decision making (cf. definition of the term autonomous [COED09], and definitions of autonomous applications [BeHP]).

Such applications are autonomous in the sense that the only way to access their confidential data is by asking an application to perform a task on the application’s data and return the output to the requester [SuSu03]. For example, a requester (an application or a person) may request an autonomous application: “Display the value of your data item X,” and the autonomous application displays a value of X. An autonomous application may use services of other applications or systems—such as the functions of an operating system. These services may provide the requested functionalities but do not make decisions for the autonomous applications.

Autonomous applications have a wide range of uses in many areas, including service oriented architecture, pervasive computing, and robotics (e.g., autonomous vehicles).

In the context of this research, it is important to note that mobile agents are a subclass of autonomous applications.
We assume here that an autonomous application is a *single* mobile agent (see Chapter 5 for details on mobile agents), a *single* service or a *single* standalone application. In the future, we might consider autonomous applications that include *multiple* mobile agents, services, or standalone applications.

An autonomous application has an *owner*—a human or artificial entity who creates them (or paid for their creation) and on whose behalf they operate.

Untrusted hosts executing autonomous applications are a threat to the applications since they have full access to the applications’ confidential data and code.

In Chapter 6 we described the ABTTP prototype implementing the active bundle scheme using a Trusted Third Party (TTP). However, a TTP, being a centralized solution, has its own issues. For instance, a TTP is prone to attacks. This led us to think about investigating approaches for implementing the active bundle scheme without the use of a TTP. More specifically, to investigate implementing active bundles as autonomous applications (see Subsection 1.4 for the discussion about the different implementations of the active bundle scheme). The goal for investigating protecting confidentiality of autonomous applications using secure computing is: *Find out whether it is possible to use secure computing to protect autonomous applications.*

If a positive answer to this question is obtained, we will be motivated to implement the active bundle scheme as an autonomous application.

Note that this chapter is extracted from [BLGT10].
7.1.1. Characteristics of Autonomous Applications

Autonomous applications have two basic characteristics: computation autonomy and data autonomy. An autonomous application exhibits *computation autonomy* if and only if its entire code is computed by the visited host alone (and all decisions necessary to execute this code completely require no interaction with other hosts).

A visited host running a *computation-autonomous application* may request and use data (e.g., input, decryption keys) from other parties while computing the agent’s code. The following property disallows such transfer of data from third parties. An autonomous application exhibits *data autonomy* if and only if it uses only data that it or its visited host possesses (e.g., its own input, its host’s input, decryption keys created or already possessed by it, or decryption keys created or already possessed by its host).

Thus, a host computing an autonomous application does *not* ask for or receive any data from any other party while computing the application’s code (cf. Subsection 5.4 for more details).

7.1.2. Selected Approaches for Protecting Autonomous Applications

Recall that *secure computing* is computing that does not allow for any unauthorized disclosure of data or functions (cf. Section 3.1).

Secure computing has been proposed in the literature to protect confidentiality of mobile agents (which, as assumed above, are the subclass of autonomous applications). However, we have already shown (in Chapter 5) that there are no *known* solutions for protecting confidentiality of mobile agents in cases when the agents provide output to
visited malicious hosts. Note that no results about theoretical feasibility of such solutions, specific enough for our needs, are available.

As discussed in Chapter 3, secure computing approaches include secure computing with encrypted data, and secure computing with encrypted functions (the third category, not used by us in this chapter, is secure computing with encrypted data and encrypted functions).

Secure computing with encrypted data can be implemented, e.g., using a method known as privacy homomorphism (cf. Chapter 3). Privacy-homomorphic computing\(^48\) for a program \(P\) is executing \(P\) while input data used by \(P\) are encrypted using a privacy homomorphism.

Secure computing with encrypted functions can be implemented, e.g., using a method known as garbled circuits [Yao86] (cf. Chapter 3). However, this chapter considers a generic method that implements secure computing with encrypted functions, not a specific one such as garbled circuits.

We investigate here two alternative approaches for protecting autonomous applications:

1) Protecting confidentiality based on secure computing with encrypted data (CED)—using privacy homomorphisms (discussed in Section 7.4).

2) Protecting confidentiality based on secure computing with encrypted functions (CEF)—using generic encrypted functions (discussed in Section 7.5).

\(^{48}\) This is an established name [RAD78, FeHe98, Ferr96] but a more precise name would be confidentiality-homomorphic computing.
7.1.3. SAS Computing and SAS Virtual Machine

We have already defined autonomous applications. Similarly, computing is autonomous when it is able to complete successfully without interacting with any other party (e.g., this application owner, any other user, another application, etc.).

Computing is sequential when it executes code sequentially (e.g., one code line at a time); otherwise, it is parallel.

A virtual machine (VM) is as a collection of programs that, when executed, provides a set of services that other software can utilize without knowing how those services are implemented [BBCC00]. We use the acronym SAS to stand for secure, autonomous, and sequential. For example, a SAS VM is a VM that realizes SAS computing.

7.1.4. Chapter Contributions and Organization

Assume that we have a virtual machine that computes Program $P$. This chapter investigates the following two questions:

1) Given a program (function) $P$ executed via privacy-homomorphic secure computing with encrypted data (a method that implements CED), Is it possible to have a SAS VM executing $P$ that provides the output of $P$ as a plaintext to a visited host? (We are not interested in either existence of a non-SAS VM or existence of a SAS VM that does not provide the output of $P$ as a plaintext to a visited host.)

2) Given a program (function) $P$ and program encryption (encrypted function) $E(P)$ executed via secure computing with the generic encrypted function (a method that implements CEF), is it possible to have a SAS VM executing $E(P)$ that provides the
output of \( E(P) \) as a plaintext to a visited host? (Again, we are not interested in either existence of a non-SAS VM or existence of a SAS VM that does not provide the output of \( P \) as a plaintext to a visited host.)

Our main contribution is proving theorems that answer both questions negatively. In terms of a SAS-based autonomous application (as used in the two questions above), we prove that:

1) Given a program (function) \( P \) executed via privacy-homomorphism (a subcategory of CED), there is no SAS VM executing \( P \) that provides the output of \( P \) as a plaintext to a visited host.

2) Given a program (function) \( P \) and program encryption (encrypted function) \( E(P) \) executed via secure computing with the generic encrypted function (a subcategory of CEF), there is no SAS VM executing \( E(P) \) that provides the output of \( E(P) \) as a plaintext to a visited host.

The chapter is organized as follows. Section 7.2 discusses the related work. Section 7.3 presents our approach to protecting confidentiality for autonomous applications using SAS computing (secure, autonomous and sequential computing). Sections 7.4 and 7.5 describe two approaches for protecting confidentiality of autonomous applications. The first approach is privacy-homomorphism. The second approach is secure computing with the generic encrypted function. Section 7.6 discusses the obtained results. Section 7.7 concludes the chapter and proposes future extensions of this work.

Note that the result in Section 7.4 is for the universal privacy-homomorphic decryption function \( D_s \) while the nonexistence result in Section 7.5 is for the universal encrypted decryption function \( D_s \).
7.2. Related Work

Approaches for protecting confidentiality of autonomous applications include the following four categories: (i) program obfuscation, (ii) using secure co-processors, (iii) secure computing with encrypted functions using cryptocomputing, and (iv) computing fixed function using a Trusted Platform Module (TPM).

7.2.1. Program Obfuscation

Obfuscation refers to hiding data and real program code in the plaintext form within a scrambled code (cf. [CFT03]). The goal of the research on program obfuscation is to develop means to scramble code so that it still works, but provably cannot be reverse engineered [Hoh09]. Several solutions have been proposed in the literature for the use of program obfuscation to protect confidentiality of autonomous applications such as autonomous mobile agents. Sarmenta [Sarm99] discusses the use of obfuscation for protecting a program from hostile environments; this applies also to autonomous applications. D’Anna et al. [AMRV03] investigate the use of obfuscation for self-protecting mobile agents.

7.2.2. Secure Coprocessors

The use of secure co-processors for protecting confidentiality of autonomous applications was proposed, for example, by IBM. IBM 4758 PCI Cryptographic
Coprocessor [IBM09] enables performing sensitive processing, and includes a secure computing environment for the storage of keys.

This approach has three main disadvantages: (i) a coprocessor can execute only programs that are encrypted with one of the keys that it holds in the key store; (ii) an operating system can leak information from an encrypted program when it calls a procedure from an unencrypted library (since the calling procedure provides plaintext parameter values to the called procedure [LHH03]); and (iii) the approach requires additional hardware.

Due to its disadvantages, we exclude the secure coprocessors approach from our further considerations.

7.2.3. Secure Computing with Encrypted Functions Using Cryptocomputing

Sander and Young [SaYo99] propose a method, named by them cryptocomputing, for secure computing with encrypted functions. They define cryptocomputing as computing that allows for the non-interactive execution of “encrypted programs” [SaYo99]. “Encrypted programs” include programs that use encrypted data and programs whose functions (represented by circuits) are encrypted. Non-interactive computing refers to computing functions without interacting with any other party except to get the input to the program.
Sander and Tschudin [SaTs98b] discussed the use of *universal Boolean functions*\(^{49}\) for cryptocomputing. Sander and Young’s idea is to consider a circuit \(C\) that realizes a function \(f\) as the input \(I_f\) for the *universal circuit* \(U\). The output of the universal circuit \(U\) is computed using its input \(I_f\), as well as input data for \(f\), if any are provided.

Let \(f\) be any function of the *family of Boolean functions* with the input size \(n\),\(^{50}\) and let \(m\) be the size of the input \(I_f\) for the universal circuit \(U\) (which happens to be equal to the size of \(U\)). For \(U\) to be a universal circuit for function \(f\), \(m\) must be at least \(2^n/\log_2 (2n+2)\) [SaTs98b, Pre70].

Sander and Young [SaYo99] provide a bound for the size and depth\(^{51}\) of a universal circuit that can compute the family of functions \(f\). Let \(s\) and \(d\) be, respectively, the size and depth of circuit \(C\) that realizes \(f\). The size and depth of a universal circuit \(U\) that can compute circuit \(C\) is on the order of \(O(d \times s \log s)\) and \(O(d \log s)\), respectively.

### 7.2.4. Computing of Fixed-functions using Trusted Platform Module

Trusted Platform Module (TPM) provides memory and a set of operations that can be performed on it with primitives such as decrypting data and signing data. The operations are not sufficient for performing arbitrary computations on TPM. Instead, TPM was

---

\(^{49}\) Let \(U(z_1, z_2, \ldots, z_m)\) be a Boolean function with \(m\) variables (i.e., the size of its input). Let \(f(x_1, x_2, \ldots, x_n)\) be a Boolean function with \(n\) variables. The *Boolean function* \(U\) is *universal* if it realizes all Boolean functions \(f\) by substituting for each \(z_i\) (input to \(U\)), a variable from the set \(\{0, 1, x_1, \ldots, x_n, x_1, \ldots, x_n\}\) [Pre1970].

\(^{50}\) Families of Boolean functions are determined here by the size of their input.

\(^{51}\) The *depth* of a circuit \(C\) is the maximum number of gates required to generate an output using an input to \(C\) (cf. [Saf2008]).
envisioned to only attest that a host being verified (to which TPM is attached) runs a trusted software stack (trusted by its owner) [CSDD08]. TPM can compute a set of specific functions (i.e., fixed-functions) (not arbitrarily functions) which includes decrypting data, and signing data.

7.3. Proposed Approach to Protecting Confidentiality for Autonomous Applications Using SAS Computing

A layer is a partition of an application such that each partition can be viewed as a virtual machine [BBCC00]. An autonomous application can be divided into many layers. For example, an application can include a (hardware) platform layer, a communication layer, an application support layer, and an application subsystem layer [BBCC00]. Since our focus is not on the organization and structure of software but on investigating security issues in autonomous applications, we regroup the set of computation layers that compose an application into just one layer which we name the encrypted computation superlayer.

We propose to investigate an alternative approach to protecting confidentiality of autonomous applications, based on having the following two superlayers:

1) Encrypted computation superlayer—It encapsulates all computations of an autonomous application (as mentioned in the preceding paragraph).

2) SAS VM superlayer—It is a SAS VM that “simulates” (i.e., executes) the application (i.e., supports the encrypted computation superlayer).

The two superlayers together cover all computations included in an autonomous application (covered by the encrypted computation superlayer) as well as mechanisms
protecting confidentiality of this autonomous application. Fig. 7.1 shows the superlayer structure of a SAS-based autonomous application, as proposed by us.

One of the properties of autonomous applications, as defined in this chapter, is that they execute on a single host (thus assuring computation autonomy).\textsuperscript{52} To be secure, an autonomous application must protect its own confidentiality. Any approach for providing security to autonomous applications must consider the following challenges: (i) an adversary has complete control over the computing host (or the host itself is an adversary); and (ii) an autonomous application cannot rely on other parties—even for providing security. This excludes the use of TTPs (such as, e.g., use of TTPs for mobile agent security [ACCK01]).

\begin{figure}[h]
\centering
\begin{tabular}{|c|}
\hline
Encrypted computation superlayer \\
\hline
SAS VM superlayer \\
\hline
\end{tabular}
\caption{The superlayer structure of a SAS-based autonomous application}
\end{figure}

As a corollary of our definition of computation autonomy, any application that requires more than one host for completing its computation is a non-autonomous application. Multiple hosts participating in computing a non-autonomous application may communicate and collaborate to execute programs that constitute the complete computation of the application. Confidentiality of computing non-autonomous

\textsuperscript{52} Recall our assumption that an autonomous application is a single mobile agent, a service or a standalone application. Hence, its computations are run completely on a single host.
applications can be protected by using, for example, secure multi-party computing protocols [BMR90]. This issue is beyond the scope of this Thesis.

For both autonomous and non-autonomous applications, computations and security services can be provided by a single superlayer or multiple superlayers, as shown in Table 7.1 (non-autonomous applications are shown for completeness.)

Table 7.1 shows the number of hosts needed for performing computations for autonomous (and non-autonomous) applications as the function of the number of layers that implement the application. “1 host” means that computing for an autonomous application must be completed by only one host at a time (“at a time” because autonomous applications can migrate over time from one host to another host). This means that the host must not communicate with any other host during computations. (For completeness, we note that computing for a non-autonomous application might require more than one host; to assure secure computing in this case, multiple hosts participating in computing have to cooperate to preserve confidentiality of their collaborative computations.)

In the remainder of this chapter we investigate whether it is possible to construct a SAS VM using a single-host autonomous application consisting of the two identified superlayers (the encrypted computation superlayer and the SAS VM superlayer).

<table>
<thead>
<tr>
<th>Number of superlayers</th>
<th>Single superlayer</th>
<th>Multiple superlayers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autonomous applications</td>
<td>1 host</td>
<td>1 host</td>
</tr>
<tr>
<td>Non-autonomous applications</td>
<td>1 host</td>
<td>1 or many hosts</td>
</tr>
</tbody>
</table>
7.4. **Proof of Nonexistence of a Universal Privacy-homomorphic Decryption Function**

This section investigates the following: given a program (function) $P$ executed via privacy-homomorphic secure computing with encrypted input data, there is no SAS VM executing $P$ that provides the output of $P$ as a plaintext to a visited host.

Recall that a *universal* privacy-homomorphic decryption function is a privacy-homomorphic decryption function that can decrypt *any* function. This section presents our results on the nonexistence of a universal privacy-homomorphic decryption function $D_g$ for decrypting an arbitrary sequential function $g$ to be protected.

Recall from Chapter 3 that a privacy-homomorphism must have the following property (since it is a method for computing with encrypted data):

$$D( f(E(x)) ) = f( D(E(x)) ) = f(x)$$  \hspace{1cm} (7.1)

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</td>
<td>a one-argument evaluation function</td>
</tr>
<tr>
<td>$g(y, z)$</td>
<td>a one- or two-arguments function to be protected (the two-arguments function is shown here); the reason for having two arguments is that the function supports also the case where the function $g$ is to be computed for two parties (e.g., the owner of $g$ and the host of $g$), and each party gives its own argument to the function;</td>
</tr>
<tr>
<td>$E_f/E_g$</td>
<td>encryption function for function $f / g$</td>
</tr>
<tr>
<td>$D_f/D_g$</td>
<td>decryption function for function $f / g$</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>the set of sequential functions</td>
</tr>
<tr>
<td>$v \circ u$</td>
<td>composition of functions $v$ and $u$</td>
</tr>
</tbody>
</table>
The following proof shows that it is not possible to have a universal privacy-homomorphic decryption function. Table 7.2 provides notation that we use in this section.

Referring to Fig. 7.1, showing the structure of a SAS-based autonomous application, we can say that: (i) \( g \) is the function to be encrypted and calculated within the encrypted computation superlayer, and (ii) \( f \circ E_f \) realizes the functionality of the SAS VM (thus, it constitutes the SAS VM superlayer).

Recall that the above encryption/decryption functions have the following properties:

\[
D_f(E_f(x)) = x \tag{7.2}
\]

\[
D_g(E_g(x)) = x \tag{7.3}
\]

**Definition 7.1.** [Skr09] Two functions \( f \) and \( g \) are *equal functions*, denoted \( f = g \) iff:

(i) \( \text{dom } f = \text{dom } g = X \)

(ii) \( f(x) = g(x) \) for each \( x \in X \)

(iii) \( \text{codom } f = \text{codom } g \)

where \( \text{dom} \) and \( \text{codom} \) refer to domain and co-domain, respectively.

**Definition 7.2.** Let \( f \) be a function representing the VM, \( x \)—an encrypted input for the function, and \( y \)—the plaintext output of the function. The privacy-homomorphic function representing a SAS VM is defined as \( y = f(E_f(x)) \).

Note that if we apply \( f \) without \( E_f \), then the plaintext output \( z \) satisfies the equation \( z = f(g(E_g(x), E_g(y)) = D(E_g(x), E_g(y)) \). In this case \( f \) is equivalent to the decryption function (based on Eq. 7.1) that uses the decryption key associated with encryption key used by \( E_g \). (Recall that for any cryptosystem, the encryption key and decryption key must exhibit
relation of Eq. 7.2.) However, if we apply $f$ with $E_f$, then we have a privacy homomorphism that uses its own encryption function and encryption key.

Note that the proofs in this chapter do not explicitly use the fact that functions $f$ and $g$ are sequential. However, we are limiting our claims to sequential functions since we are not sure that all steps of the proofs apply also to parallel functions. (For instance, we know that Eq. 7.9 below is valid for sequential functions but we are not confident if it is true for parallel functions.)

**Lemma 7.1.**

Let $g(x, y)$ be an arbitrary sequential function to be protected using secure computing with encrypted data, $E_g/D_g$ be the encryption/decryption function for $g$ (used to homomorphically encrypt/decrypt $g$), $f$ be a function that implements SAS VM, and $E_f$ be the encryption function for $f$ (used by $f$ in the privacy-homomorphic cryptosystem to homomorphically compute function $g$).

The application of privacy-homomorphic encryption function $E_f$ for any input $h = E_g(g(x,y))$ followed by application of function $f$ is equal to the application of $D_g$ to this input.

Let $g(x, y)$ be an arbitrary sequential function to be protected using secure computing with encrypted data CED;

$E_g/D_g$ the encryption/decryption functions for $g$ (used to privacy-homorphically encrypt/decrypt $g$), respectively; $f$ a SAS function that implements SAS VM; and $E_f$ the encryption function for $f$ (used to encrypt input to $f$). Then:

$$g(x, y) \in \mathcal{X} \text{ and } h(x, y) = E_g(g(x,y)) \implies D_g(h) = f \circ E_f(h)$$
This means that applying the privacy-homomorphic encryption function $E_f$ for any input $h = E_g(g(x, y))$ followed by application of sequential function $f$ is equal to the application of $D_g$ to this input.\(^{53}\)

**Proof:** Let $g(x, y)$ be a multivariate sequential function (its input, in this case, are two variables).

Let

$$z = g(x, y) \quad (7.4)$$

Applying Eq. 7.1 to function $g$ gives:

$$g(x, y) = D_g(g( E_g(x), E_g(y) ) ) \quad (7.5)$$

By combining Eq. 7.4 and Eq. 7.5 we get:

$$z = D_g( g( E_g(x), E_g(y)) ) \quad (7.6)$$

Definition 7.2 says that:

$$m = f( E_f(n)) \quad (7.7)$$

Let the result of computing $g(E_g(x), E_g(y))$ be the input to SAS VM, $n$; that is, let:

$$n = g(E_g(x), E_g(y)) \quad (7.8)$$

Observe that the value $n$ is computed here using privacy homomorphism since $g$ computes with encrypted data.

Eq. 7.6 and Eq. 7.8 give:

$$z = D_g(n) \quad (7.9)$$

\(^{53}\) Note that the composite function $f \circ E_f$ uses a privacy homomorphism to compute $g$, and provides the output as a plaintext
Computing \( g \) with encrypted input, SAS VM must provide the same output for \( g \) as it provides when computing \( g \) with plaintext input. This means that:

\[
m = z \quad (7.10)
\]

Combining Eq. 7.9, Eq.7.10, and Eq.7 gives:

\[
D_g(n) = z = m = f( E_f(n) ) = f \circ E_f(n) \quad (7.11)
\]

Since function \( D_g \) and function \( f \circ E_f \) use the same input \( n \), they have the same domain and co-domain. Therefore by Definition 7.1, Eq. 7.11 gives:

\[
D_g = f \circ E_f \quad (7.12)
\]

Eq. 7.12 shows that the decryption function for the function \( g \) is equal to function \( f \) composed with the encryption function \( E_f \).

Q.E.D.

We have shown in Section 7.3 the proposed superlayer structure of a SAS-based autonomous application.

Functions \( E_g/D_g \) are used to encrypt input/decrypt output of \( g \) within the deployed privacy-homomorphism (implementing the CED approach). Functions \( f \) and \( E_f \) define SAS VM, where \( E_f \) is the encryption function for the privacy homomorphism used by function \( f \). Function \( g \) defines the autonomous application, and \( D_g \) is the decryption function used by \( g \). Hence, Lemma 1 simply states that \( D_g \) is equal to the composition of \( E_f \) and \( f \), that is, \( D_g \) is equal to the function representing SAS VM (because SAS VM is a composition of \( E_f \) and \( f \)). More specifically, \( D_g \) applied to \( h(x, y) \) produces the same output as SAS VM applied to \( h(x, y) \).
Theorem 7.1.

Let $g$ be an arbitrary sequential function to be protected, and $f$ be a sequential privacy-homomorphic simulation function\(^{54}\) with its encryption function $E_f$. Assume that decryption functions for two different functions $g_1$ and $g_2$ are different. Then, there is no universal privacy-homomorphic decryption function $D_g$ (when both $f$ and $g$ are sequential, as demanded).

Proof: We prove this theorem by contradiction. Let $g_1$ and $g_2$ be two different (hence, unequal) sequential functions, such that:

$$z = g_1(x, y) \quad (7.13)$$

and

$$w = g_2(x, y) \quad (7.14)$$

Assume that there exists a universal privacy-homomorphic decryption function $D_g$ for a given sequential privacy-homomorphic simulation function $f$ and its encryption function $E_f$.

Let $D_{g_1}$ and $D_{g_2}$ be the decryption functions for $g_1$ and $g_2$, respectively. Both $g_1$ and $g_2$ are to be computed by SAS VM, represented by $f \circ E$.

Applying Lemma 7.1 to Eq. 7.13 and Eq. 7.14 in turn gives:

$$D_{g_1} = f \circ E_f \quad (7.15)$$

$$D_{g_2} = f \circ E_f \quad (7.16)$$

Then, combining Eq. 7.15 and Eq. 7.16 gives

---

\(^{54}\) A simulation function is a function that executes (evaluates) another function.
\[ D_{g2} = D_{g1} \]  \hspace{1cm} (7.17)

Eq. 7.17 contradicts the assumption of the theorem, which states that “\textit{decryption functions associated with two different sequential functions are different.}” This means that the proof assumption stating: “\textit{there exists a universal privacy-homomorphic decryption function } D_{s} \text{ for a given sequential privacy-homomorphic simulation function } f \text{ and its encryption function } E_{f}’’ \textit{leads to a false conclusion, thus the proof assumption is not true. Hence, there is no universal privacy-homomorphic decryption function } D_{g}. \text{ Q.E.D.}

We have shown in Section 7.3 our proposed structure of secure autonomous applications. Functions \( f \) and \( E_{f} \) are the formulation of SAS VM where \( E_{f} \) is the encryption function used by function \( f \). Function \( g \) is the formulation of an autonomous application, which uses \( D_{s} \) as the decryption function for \( g \).

In terms of a SAS-based autonomous application, the theorem states that we cannot have a universal privacy-homomorphic decryption function that could be implemented as a SAS VM (represented by function \( f \circ E_{f} \)).

A result of Theorem 7.1 is that \( f \circ E_{f} \) has a relationship with \( g(E_{g}(x), E_{g}(y)) \) and therefore both functions are dependents. If a universal privacy-homomorphic decryption exists then both functions must be independent.

As an example of using the results of this theorem, it would prevent wasting resources in futile attempts to build a SAS VM that uses privacy homomorphisms, and which would be used in cloud computing to simulate (execute) programs that use privacy homomorphisms.
7.5. **Proof of Nonexistence of a Universal Encrypted Decryption Function**

This section investigates the following: Given a program (function) $P$ and program encryption (encrypted function) $E(P)$ executed via secure computing with the *generic* encrypted function there is no SAS VM executing $E(P)$ that provides the output of $E(P)$ as a plaintext to a visited host.

We present in this section our result on the nonexistence of the universal *encrypted* decryption function $D_g$ for an arbitrary sequential function $g$ (the sequential evaluation function $f$ together with its encryption function $E_f$ correspond to SAS VM).

Referring to the structure of a SAS-based autonomous application (cf. Fig. 7.1), we can say that $g$ is the function to be calculated using its encrypted function $E_g$, and $E_f \circ f$ constitutes the SAS VM.

Recall from Section 3.2.2 that a protocol for secure computing with encrypted functions (CEF) must have the following property:

$$D(E(f)(x)) = f(x) \quad (7.18)$$

**Definition 7.3.** Let $f$ be a function, $n$ — an encrypted input for function $f$, and $m$ — the plaintext output of function $f$ for input $n$.

Since function $f$ and its encryption $E_f$ together represent a SAS VM, a computation performed by SAS VM can be represented as $m = E_f(f(n))$.

Note that in Definition 7.2 $f$ is computed with encrypted data and in Definition 7.3 encrypted function $f$ is computed using encrypted function. Note also that $E_f$ is an encryption for functions in this section while it is an encryption for data used by function $f$ in Section 7.4.
Note that if we apply $f$ without $E_f$, then the plaintext output $z$ satisfies the equation $z = f(E_g(g)(x,y)) = D(E_g(g)(x,y))$. In this case $f$ is a decryption function (based on Eq. 7.1) that uses the decryption key associated with the encryption key of $E_g$. However, if we apply $f$ with $E_f$ then we have an encrypted function that uses its own encryption function and encryption key.

**Lemma 7.2**

Let $g(x, y)$ be an arbitrary sequential function to be protected using secure computing with encrypted functions $CEF$, $E_g/D_g$ — the encryption/decryption function for $g$, $f$ — a SAS function that implements SAS VM, and $E_f$ — the encryption function for $f$.

Then:

$$g(x, y) \in \mathcal{E} \text{ and } h(x, y) = E_g(g)(x,y) \Rightarrow D_g(h) = E_f(f)(h)$$

This means that applying the function $f$ for any input $h = E_g(g(x,y))$ followed by the application of encryption function $E_f$ is equal to the application of $D_g$ to this input.

Recall that $E_f$ is applied here to function $f$, while in Lemma 7.1 $E_f$ is applied to input data (to input $h$).

**Proof:** Let

$$z = g(x,y) \quad (7.19)$$

Applying Eq. 7.18 to function $g$ gives:

$$g(x,y) = D_g( g( E_g(x), E_g(y) ) ) \quad (7.20)$$

By combining Eq. 7.19 and Eq. 7.20 we get:

$$z = D_g( E_g(g)(x,y) ) \quad (7.21)$$

Definition 7.3 says that:
\[ m = E_f(f)(n) \]  
\hspace{1cm} (7.22)

Let the result of computing \( E_g(g)(x, y) \) be the input to the SAS VM, \( n \); that is, let:

\[ n = E_g(g)(x, y) \]  
\hspace{1cm} (7.23)

Observe that the value \( n \) is computed here using the encrypted function \( g \)

Eq. 7.21 and Eq. 7.23 give:

\[ z = D_g(n) \]  
\hspace{1cm} (7.24)

Let \( n \), the result of computing \( E_g(g)(x, y) \), be the input to the SAS VM.

Recall that \( m \) is the output when computing \( g \) with encrypted function \( E_g \) and \( E_f(f) \), and \( z \) is the output when computing \( g \) with plaintext function \( g \).

When computing \( g \) with an encrypted function, SAS VM must provide the same output for \( g \) as it provides when computing \( g \) with plaintext function. This means that:

\[ m = z \]  
\hspace{1cm} (7.25)

Substituting Eq. 7.22 and Eq. 7.24 into Eq. 7.25 we get:

\[ E_f(f)(n) = D_g(n) \]  
\hspace{1cm} (7.26)

Eq. 7.26 shows that applying function \( E_f(f) \) to \( n \) is equal to applying the decryption function \( D_g \) to \( n \). This is equivalent to saying that \( D_g(h) = E_f(f)(h) \), which completes the proof.

Q.E.D.
We have shown in Section 7.3 the proposed structure of SAS-based autonomous applications.

Functions $E_g/D_g$ are used to encrypt/decrypt $g$ within the generic CEF method (implementing the CEF approach). Functions $f$ and $E_f$ define SAS VM, where $E_f$ is the encryption function used by $f$ in the generic CEF method for computing function $g$. Function $g$ defines the autonomous application, and $D_g$ is the decryption function for the generic CEF method used by $g$. Hence, Lemma 7.2 simply states that $D_g$ is equal to the composition of $f$ and $E_f$, that is, $D_g$ is equal to the function representing SAS VM (because SAS VM is a composition of $f$ and $E_f$). More specifically, $D_g$ applied to $h(x, y)$ produces the same output as SAS VM (composition of $f$ and $E_f$ in this case) applied to $h(x, y)$.

**Theorem 7.2.**

Suppose that $g$ is an arbitrary function to be protected, and $f$ is the simulation function. Assume that both $f$ and $g$ are sequential. Assume that the decryption functions associated to two different sequential functions $g_1$ and $g_2$ are different. Then, there is no universal encrypted decryption function $D_g$.

**Proof:** We prove the theorem by contradiction. Let $g_1$ and $g_2$ be two sequential functions.

Let $D_{g_1}$ and $D_{g_2}$ be the decryption functions for $g_1$ and $g_2$, respectively. Both $g_1$ and $g_2$ are to be computed by SAS VM, represented by $E_f(f)$.

By Lemma 7.2:

$$D_{g_1} = E_f(f)$$

(7.27)
Applying Lemma 7.2 to Eq. 7.23 gives:

\[ D_{g2} = E_f(f) \]  \hspace{1cm} (7.28)

Then combining Eq. 7.27 and Eq. 7.28 gives:

\[ D_{g1} = D_{g2} \]  \hspace{1cm} (7.29)

Eq. 7.29 contradicts the assumption of the theorem, which states that “decryption functions associated with two different sequential functions are different.” This means that the proof assumption stating: “there is a universal encrypted decryption function \( D_g \) for a given sequential encrypted simulation function \( f \)” leads to a false conclusion, thus the proof assumption is not true. Hence, there is no universal encrypted decryption function \( D_g \).

Q.E.D.

We have shown in Section 7.3 our proposed structure for secure autonomous applications. Functions \( f \) and \( E_f \) are the formulation of SAS VM where \( E_f \) is the encryption function used by function \( f \). Function \( g \) is the formulation of an autonomous application, which uses \( D_g \) as the decryption function for \( g \).

In terms of a SAS-based autonomous application, the theorem states that we cannot have a (universal) SAS VM that uses secure computing with encrypted functions (CEF) to compute an encrypted decryption function \( D_g \).

Note that in Theorem 7.1 we have a plaintext decryption function and encrypted data, while in Theorem 7.2 we have an encrypted function (and possibly plaintext data).
Theorem 7.2 means that $E(f)$ has a relationship with $E(g)$ and therefore both functions are dependents. If a universal encrypted decryption function exists then both functions must be independent.

As an example of using the results of this theorem, it would prevent wasting resources in futile attempts to build a SAS VM that uses encrypted functions. It is waste of resource for example to attempt to build SAS VM to be used in cloud computing to simulate (execute) programs that use encrypted functions.

7.6. Discussion of the Nonexistence Results

It is widely believed that an attacker who can view the computation of a processing unit is able to understand programs being executed by it and affect their execution. For example, a cloud computing platform can understand any program it executes if the program’s output is in plaintext.

In this chapter we investigated two alternatives for protecting confidentiality of autonomous applications being executed on untrusted hosts: (i) encrypting input data of an autonomous application, and (ii) encrypting the code (“function”) of an autonomous application. Instead of using a VM that computes encrypted code (as suggested by some related work in Subsection 7.2.3), we investigate using an encrypted the VM (SAS VM).

The major results of this chapter are given by Theorem 1 and 2.

The results of Theorem 1 can be interpreted as follows. Given a program (function) $P$ executed via privacy-homomorphism with encrypted input data, there is no SAS VM executing $P$ that provides the output of $P$ as a plaintext to a visited host. Even more
simply, we can say that there is no secure,\textsuperscript{55} autonomous, and sequential VM (SAS VM) using encrypted input data, using privacy homomorphisms in its computations, and providing plaintext output to a visited host that can compute an arbitrary program.

Theorem 2 can be interpreted as follows. Given a program (function) $P$ and program encryption (encrypted function) $E(P)$ executed via secure computing with the \textit{generic} encrypted function, there is no SAS VM executing $E(P)$ that provides the output of $E(P)$ as a plaintext to a visited host. Even more simply, we can say that there is no secure, autonomous, and sequential VM (SAS VM) with an encrypted program ("function" in the CEF terms) and providing plaintext output to a visited host that can compute an arbitrary program.

The negative theoretical results for “universal” solutions for decryption functions and their practical consequences for “universal” SAS VMs reinforce our belief that it is not possible to have “universal” or “generic” solutions for protecting confidentiality of autonomous applications providing plaintext output to their visited hosts.

This means that we cannot use secure computing approaches to construct a SAS VM with encrypted input data or encrypted code that could be included in active bundles. However, the potential solution space is not nullified by these results. The combined consequences of Theorem 1 and 2 do not exclude the possibility that there exist: (a) \textit{non}-universal (i.e., domain specific) privacy-homomorphic decryption functions (by Theorem 1); (b) \textit{non}-universal (i.e., domain specific) encrypted decryption functions (by Theorem 2); and (c) universal decryption functions that are \textit{neither} privacy-homomorphic \textit{nor} encrypted functions (by Theorems 1 and 2). Finally, all these results were obtained for

\textsuperscript{55} Recall that “secure” means here “protecting confidentiality.”
sequential functions, so Theorems 1 and 2 do not exclude the possibility that there exist: (d) universal privacy-homomorphic parallel decryption functions; and (e) universal encrypted parallel decryption functions. These five “openings” dictate the available directions for promising future research.

We believe that the main cause of the difficulty in developing a solution for protecting confidentiality of autonomous applications providing plaintext output to their visited hosts is that most programs are deterministic (i.e., given a program and an input to the program, it is possible to predict the output with probability 1). An attacker who gets an input and a nondeterministic program can not predict the output of the program since the program outputs a different element from a set of possible values each time it is executed. Randomized programs (RPs), or probabilistic programs (PPs) are two categories of nondeterministic programs [ArBa09]. (Arora and Barak used the terms “randomized algorithms” and “probabilistic algorithms” [Arba09]. Since an algorithm is implemented as a program, then we extend these terms to “randomized programs” and “probabilistic programs.”) Both program categories use a degree of randomness in their logic. If there is a transformation that can transform a deterministic program to a nondeterministic program while preserving its functionality, then such transformation could be used to obfuscate the deterministic program. Therefore, the transformation could be used to obfuscate VM’s code included in active bundles.
7.7. Conclusions

This chapter presented an approach for protecting confidentiality of an arbitrary autonomous application when the application’s (input) data or code is encrypted, and a secure autonomous sequential VM (SAS VM) simulates (evaluates) the application.

The major contributions of this chapter are: (i) proving the nonexistence of a universal privacy-homomorphic decryption function; (ii) proving the nonexistence of a universal encrypted decryption function. The positive consequence of these negative results is the identification of promising venues or “openings” for further research.

Encouraged by the five “openings” identified in the previous sections, we raise the following question: How to protect confidentiality of autonomous applications using an approach different than using secure computing for protecting VM data (CED) or VM code (CEF)? In our first attempt to answer this question, Chapter 8 investigates the use of secure computing for protecting confidentiality of control flow graphs for VMs’ code (rather than for protecting confidentiality of the VMs’ code “directly,” i.e., not via protecting their control flow graphs). Specifically, we propose obfuscating control flow graphs for VMs’ code.
CHAPTER 8

PROTECTING CONFIDENTIALITY OF AUTONOMOUS APPLICATIONS WITH OBFUSCATED CONTROL FLOW GRAPHS

8.1. Introduction

Obfuscation refers to hiding information (data and code) in plaintext within scrambled computer code [CFT03].

The obfuscation properties are [Hoh07]:

1) Preserving functionality of the original program, i.e., the target (obfuscated) program should produce exactly the same output as the source program.

2) The target code can suffer at most a polynomial slowdown compared to the source program.

3) The target code is a “virtual black box,” i.e., it does not leak any information to would-be reverse engineers.

Another widely used definition of obfuscation says that “an obfuscator $O$ is an (efficient, probabilistic) ‘compiler’ that takes as input a program $P$ and produces a new program $O(P)$ that has the same functionality as $P$ yet is intelligible in some sense” [BGIR01].

Recall that Barak et al. [BGIR01], based on their definition of obfuscation formalizing the obfuscation properties described above, prove that it is not possible to have obfuscation that works for any program (a.k.a. functions). Their results do not exclude the
possibility that there are obfuscations for some families of programs, such as point functions or deterministic finite automata [KSVZ07].

Many researchers in the area of obfuscation propose obfuscating control flow graphs of programs rather than obfuscating programs themselves [COTH09]. Most solutions for program obfuscation are actually obfuscations of control flow graphs (CFGs) of their programs (i.e., changing the paths of the control flow graph in the program to be obfuscated). Examples of transformations are: inserting sets of statements, adding branching statements, and adding loops. Taxonomy of these obfuscating program transformations is discussed by Collberg et al. [COTH09]. Also, Tsai et al. discuss [THW09] a set of control flow graph transformations for obfuscations, and measure their security.

The main goals in investigating program obfuscation are: (1) finding a transformation (or many) of any program to another program such that the transformed (i.e., “obfuscated”) program preserves the functionalities of the original program; and (2) finding a transformation that assures the first goal and also assures that it is difficult to reverse engineer the obfuscated program. In this work we achieve the first goal by providing Theorem 8.1. We discuss the second goal in Section 8.6.

This chapter is organized as follows. Section 8.2 discusses control flow graphs. Section 8.3 presents our proposed solution for obfuscating control flow graphs. Section 8.4 investigates protecting confidentiality of programs using obfuscated control flow graphs and privacy homomorphisms. Section 8.5 investigates protecting confidentiality of programs using obfuscated control flow graphs and generic encrypted functions. Section 8.6 discusses the results, and Section 8.7 concludes the chapter.
This chapter is extracted from our published paper [BLGT10].

8.2. Control Flow Graphs

A program is composed of a set of statements. A statement is either a branching statement (e.g., a loop statement, an if statement, or a goto statement) or a sequential statement (i.e., a non-branch statement). Two statements are connected if and only if the second statement could be executed immediately after the first statement.

Table 8.1 describes the notation used in this section.

Definition 8.1 (cf. [HeUl74, THW09]) A control flow graph (CFG) is a directed graph represented as the triple \( G = (N, \Lambda, n_0) \), where:

1) \( N \) is a finite set of nodes.
2) \( \Lambda \subset N \times N \) is the set of edges. An edge \((n_1, n_2)\) leaves node \( n_1 \) and enters node \( n_2 \).
3) Node \( n_0 \in N \) is the initial node for CFG \( G \). \( G \) includes a path from \( n_0 \) to every node in \( N \).

A CFG represents the flow of control between statements [ALSU07]. Statements are represented in a CFG as nodes, and the flow of control between two statements is represented by an edge. Fig. 8.1 shows an example Program \( \Pi \), and Fig. 8.2 shows the control flow graph \( G \) for \( \Pi \), with branching statements labeled \( l_5 \) and \( l_7 \), and sequential statements labeled \( l_4, l_6, l_8, l_{10}, and l_{12} \).

Before providing a few definitions related to CFGs, we need to introduce notation required in this section.
<table>
<thead>
<tr>
<th>$s_i$</th>
<th>A statement with index $i$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_i$</td>
<td>The label of statement $s_k$. (most often $i \neq k$). It specifies the position of the statements in a program.</td>
</tr>
<tr>
<td>$b_i$</td>
<td>The branching function defined at position $l_i$.</td>
</tr>
<tr>
<td>$m_i$</td>
<td>The memory state before executing the statement labeled (with label) $l_i$. It includes values of all variables used by the program after executing the statement with label $l_i$.</td>
</tr>
<tr>
<td>$c_i$</td>
<td>A predicate that returns the result (either “True” or “False”) of comparing the values of specific variables (specified in the predicate) of the memory state $m_i$ to predefined values for these variables as specified in the predicate\textsuperscript{56}.</td>
</tr>
<tr>
<td>$\Pi_1$</td>
<td>An original program to be transformed</td>
</tr>
<tr>
<td>$\Pi_2$</td>
<td>The target (transformed) program</td>
</tr>
<tr>
<td>$G_1$</td>
<td>CFG of Program $\Pi_1$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>Set of paths of $G_1$</td>
</tr>
<tr>
<td>$G_2$</td>
<td>CFG of Program $\Pi_2$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>Set of paths of $G_2$</td>
</tr>
</tbody>
</table>

\textsuperscript{56} As an example, in Fig. 8.2, the statement at label $l_7$ uses predicate $c_7$ which is “$b < 80$” where “$b$” is a variable whose value is in the memory state $m_i$. $c_7$ compares the value of variable $b$ to “$80$” and outputs the result.
l₁: int test(int a, b)
l₂: {
l₃:     int c, result;
l₄:     a = 1;
l₅:     while (a < 10) {
l₆:         b = a * a;
l₇:         if (b < 80) {
l₈:             c = b + a;
l₉:         }
l₁₀:         a = a + 1;
l₁₁:     }
l₁₂:     return c;
l₁₃: }

Figure 8.1. A simple program $Π$ (in language C) that uses branching and loops

![Control Flow Graph](image)

Figure 8.2. CFG for Program $Π$

Note that a statement, say $s_i$, may be labeled with two (or even more) different labels, say $l_k$ and $l_j$. 
**Definition 8.2** The *branching function* $b_i$ at the statement labeled $l_i$ for CFG $G$ is the function $b_i(l_i, m_i, c_i)$, where $l_i$ is the label of the current statement, $m_i$ is the memory state before executing the statement at label $l_i$, and $c_i$ is a predicate to be evaluated using memory state $m_i$; such that $l_j = b_i(l_i, m_i, c_i)$ is the label of the next statement to be executed in $G$. More precisely:

$$b_i(l_i, c_i, m_i) = \begin{cases} 
  l_j \text{ if } m_i = c_i \\
  l_k \text{ if } m_i \neq c_i 
\end{cases} \quad (8.1)$$

where $l_j$ and $l_k$ are two possible outputs of $b_i$.

As an example of a branching function, the branching function for the statement labeled $l_7$ in the CFG shown in Fig. 8.2 is:

$$b_7(l_7, c_7, m_7) = \begin{cases} 
  l_8 \text{ if } b < 80 \\
  l_{10} \text{ if } b \geq 80 
\end{cases}$$

Note that the branching function formulate the edges of the CFGs. Note that we consider in this Thesis the branching function as a sequential function.

**Definition 8.3** A *path* (cf. [HeUl74]) from statement $s_i$ to $s_n$ in graph $G$ is a sequence of nodes $(s_1, \ldots, s_n)$ such that edge $(s_i, s_k)$ is an edge of $G$, $1 \leq i < k \leq n$, $s_i$ is the *initial node* of graph $G$, and $s_n$ is the *final node* in the path.

Let $p = (s_1, s_2, \ldots, s_n)$ be a path in $G$. Since $p$ is a sequence of connected edges, we can also say that $p = (s_1, s_2, \ldots, s_n) = (s_1, s_2) (s_2, s_3) \ldots (s_{i-1}, s_i) (s_n, s_{n})$.

We use *labels* $(p) = (l_1, l_2, \ldots, l_n)$ to denote the sequence of labels associated with nodes of path $p = (s_1, s_2, \ldots, s_n)$, where $l_i$ is the label for $s_i$ (cf. [JaCo97]).

**Definition 8.4** let $p = (s_1, s_2, \ldots, s_n)$. Let $P$ be the set of paths of CFG $G$. $p \in P$ iff all edges of $P$ are edges of $G$. 

177
Note that the representation of a path of a sequence of nodes in definition 8.4 assures that the edges are connected.

**Definition 8.5** Let $P_1$ be the set of all paths of graph $G_1$ and $P_2$ be the sets of all paths of graph $G_2$. $G_2$ covers $G_1$ (denoted by $G_2 \approx G_1$) iff for every path $p_1 \in P_1$ there is a path $p_2 \in P_2$, such that $p_1 = p_2$ and initial node $s_0^1$ of $p_1$ in $G_1$ is the same as the initial node $s_0^2$ of $p_2$ in $G_2$.

Formally:

$$G_1 \approx G_2 \iff (\forall p_1 \in P_1) (\exists p_2 \in P_2) : p_1 = p_2 \land s_0^1 = s_0^2$$

Note that a $p_1$ and $p_2$ have the same sequence of statements but may have a different sequence of labels.

In this work we are interested in program obfuscation using control flow graphs. In order to preserve the functionality of the original program represented by CFG $G_1$, it is sufficient to assure that $G_2$ covers $G_1$.

Now we define our obfuscation function $O$ for control flow graphs. Function $O$ is a transformation of the control flow graph $G_1$ of Program $\Pi_1$ to a control flow graph $G_2$ of the obfuscated program $\Pi_2$. 
Definition 8.6 Let $G_1$ and $G_2$ be CFGs for programs $\Pi_1$ and $\Pi_2$, respectively. We call $\Pi_2$ an obfuscation of $\Pi_1$ iff $G_2$ covers $G_1$.

$\Pi_2$ is an obfuscation of $\Pi_1$ if the program $\Pi_2$ can execute all the functions of $\Pi_1$ because all paths of its CFG, $P_1$, are in $P_2$, the set of paths of the CFG of $\Pi_2$.

8.3. Proposed Solution for Obfuscating CFGs

In this section we present our solution for obfuscating the control flow graph using secure computing approaches (i.e., CED, CEF, and CEDEF). Note that a program has two types of information that could be used for obfuscation: (i) the set of statements; and (ii) the flow of control (represented with CFGs). Our solution for obfuscating programs through obfuscating their control flow graphs is different from earlier solutions (e.g., [COTH09, JaCo97]). It is based on using encryption of data or encryption of functions of CFGs.

\[
\begin{align*}
b_1 &: a = 1 \\
b_3 &: b = a*a \\
b_5 &: c = b+a \\
b_7 &: a = a + 1 \\
b_9 &: \text{return } c
\end{align*}
\]

*Figure 8.3. Flattened program $\Pi$*

The steps of our solution for obfuscating CFGs are:

1) *Construct a branching table:* Use the CFG of the original program to develop a matrix where the rows represent statements and the columns represent the Boolean values returned by the predicates of branching functions. Table 8.1 shows the branching table for program $\Pi$. 

179
2) **Flatten the original program** \( \Pi \): Take out all statements used for the flow of control (i.e., branching statements). Fig. 8.1 shows a simple program \( \Pi \) and Fig. 8.2 shows its CFG. Fig. 8.3 shows the flattened program \( \Pi \), i.e., \( \Pi \) with all but sequential statements removed.

3) **Construct an obfuscated program**: Mix all statements of the flattened program with a set of dummy statements. Let \( k_o \) be the size of the flattened program (i.e., the number of its statements), and \( k_1 \)—the number of added extra statements. The size of the new program is \( k = k_o + k_1 \).

Note that in this Thesis we do not discuss the rules for selecting the extra statements. This task is one of our future work.

4) **Encrypt the branching table**: Encrypt one of the following: (4.i) statement labels (i.e., input and output of the branching table); or (4.ii) branching functions (i.e., the branching table itself).

Note that (4.ii) could be realized using for example a garbled circuit method, e.g., Table 3.3 is the encryption of Table 3.2.

5) **Execute the obfuscated program**: Use the encrypted branching table and the flattened program to execute the obfuscated program.

<table>
<thead>
<tr>
<th>Current statement</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_4 )</td>
<td>( l_5 )</td>
<td>( l_3 )</td>
</tr>
<tr>
<td>( l_5 )</td>
<td>( l_6 )</td>
<td>( l_{12} )</td>
</tr>
<tr>
<td>( l_6 )</td>
<td>( l_7 )</td>
<td>( l_7 )</td>
</tr>
<tr>
<td>( l_7 )</td>
<td>( l_8 )</td>
<td>( l_{10} )</td>
</tr>
<tr>
<td>( l_8 )</td>
<td>( l_{10} )</td>
<td>( l_{10} )</td>
</tr>
<tr>
<td>( l_{10} )</td>
<td>( l_5 )</td>
<td>( l_5 )</td>
</tr>
</tbody>
</table>
### Table 8.3. Notation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi$</td>
<td>Program to be obfuscated</td>
</tr>
<tr>
<td>$\Pi_1$</td>
<td>Program to be obfuscated</td>
</tr>
<tr>
<td>$\Pi_2$</td>
<td>Transformed program</td>
</tr>
<tr>
<td>$G$</td>
<td>CFG of $\Pi$</td>
</tr>
<tr>
<td>$G_1$</td>
<td>CFG of $\Pi_1$</td>
</tr>
<tr>
<td>$G_2$</td>
<td>CFG of $\Pi_2$</td>
</tr>
<tr>
<td>$\Lambda_1$</td>
<td>Set of edges of $G_1$</td>
</tr>
<tr>
<td>$\Lambda_2$</td>
<td>Set of edges of $G_2$</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>Obfuscator of programs</td>
</tr>
<tr>
<td>$O$</td>
<td>Obfuscator of CFGs</td>
</tr>
<tr>
<td>$\Psi$</td>
<td>Obfuscator of paths</td>
</tr>
<tr>
<td>$\Phi$</td>
<td>Obfuscator of edges</td>
</tr>
<tr>
<td>$E$</td>
<td>Encryption function for encrypting the branching table</td>
</tr>
<tr>
<td>$D$</td>
<td>The corresponding decryption function (for simplicity we do not specify keys for the encryption and decryption functions)</td>
</tr>
</tbody>
</table>
Note that a deobfuscator cannot reverse engineer a program whose control flow is not available. That is, a deobfuscator cannot deobfuscate a program where there is no logical sequence of the statements or no dependence between these statements.

In the following we investigate the use of computing with encrypted data (CED) and computing with encrypted functions (CEF) for obfuscating CFGs of arbitrary programs.

8.4. Protecting Confidentiality Using the Obfuscated CFGs and Privacy Homomorphisms

This section investigates the use of a privacy homomorphism to obfuscate CFGs. We encrypt the statement labels of a program $\Pi$, and compute the branching functions for the CFG representing $\Pi$. Table 8.3 describes notation used in this section.

Let $l_i$ and $l_j$ be two statement labels such that there is a branching function $b_i$ that maps $l_i$ to $l_j$, that is:

$$b_i(l_i, c_i, m_i) = l_j \quad (8.2)$$

Based on Step 4.i—encrypt the branching table through encrypting the statement labels, we obtain:

$$b_i(E(l_i), c_i, m_i) = E(l_j) \quad (8.3)$$

Eq. 8.3 is used to generate the encrypted branching table, as described in Step 4 of our solution for obfuscating CFGs. Inputs and outputs of branching functions for the CFG are encrypted and stored in the encrypted branching table. Then, we use the encrypted

---

57 Reverse engineering is the process of analyzing a program to identify its components and their interrelationships. The extracted information is used to create a representation of the system in higher level of abstraction [ChCr1990].
branching table (which represents the obfuscated CFG) to compute the obfuscated program \( \Gamma(\Pi) \) as specified in Step 5.

A privacy homomorhism can be applied to Eq. 8.2. Applying Eq. 7.1\(^{58}\) to Eq. 8.2 gives:

\[
D( b_i(E(l_i), c_i, m_i) ) = b_i( D(E(l_i), c_i, m_i) ) = l_j \tag{8.4}
\]

Note that a privacy homomorphism uses plaintext values of predicate \( c_i \) and memory \( m_i \), and encrypted values of statements labels \( l_i \) (as it is defined by the encrypted branching table).

Step 5 can be performed in two ways: (i) use the encrypted table as is (i.e., without decryption of statement labels before executing the statements), or (ii) decrypt the labels of the statements from the branching table before executing them. Both cases are analyzed next.

In the following we provide some definitions that we use in this section.

**Definition 8.7** Let \((l_i, l_{i+1})\) be an edge of \( G \). Obfuscation \( \Phi \) of \((l_i, l_{i+1})\) is the edge \((E(l_i),E(l_{i+1}))\), where \( E \) is the encryption function, \( l_i \) and \( l_{i+1} \) are two labels of nodes of \( G \).

Formally:

\[
\Phi(l_i, l_{i+1}) = (E(l_i),E(l_{i+1}))
\]

**Definition 8.8** Let \{\((l_i, l_{i+1})\)\} be a set of edges of CFG \( G \), \( p \) be a path of \( G \) where \( p = (l_{j_1},l_{j_2})..(l_{n-1}, l_n) \). Obfuscation, \( \Psi \), of \( p \) is the obfuscation of the sequence of edges of \( p \).

Formally:

\[
\Psi(p) = \Phi(l_{j_1},l_{j_2}) \ldots \Phi(l_{n-1},l_n)
\]

\(^{58}\) Recall that Eq. 7.1 is: \( D(f(E(x))) = f(D(E(x))) = f(x) \)
Case 1. Computing $O(G)$ Using the Encrypted Table As Is

When statement labels are encrypted, they have new locations in the obfuscated program (since the label of a statement is its location in the program). To shuffle the locations, an obfuscator inserts into the original program a set of extra statements. Therefore, the obfuscated program has a new CFG, $G_2$ (different from the original program’s CFG, $G_1$). The encrypted branching table of the original program becomes a part of the branching table of the obfuscated program.

In the following we investigate the use of privacy homomorphisms to obfuscate programs. First we relate the edges of a CFG to the edges of the CFG of the transformed program. Second we relate the paths of the CFG of programs to the paths of the CFG of the transformed program.

Lemma 8.1

Let CFG $G_2$ be the obfuscation of $G_1$. Let $(l_i, l_{i+1})$ be an edge of a CFG $G_1$. The obfuscation, $\Phi$, of $(l_i, l_{i+1})$ is an edge of $G_2$.

Formally,

$$(l_i, l_{i+1}) \in A_1 \Rightarrow \Phi (l_i, l_{i+1}) \in A_2$$
Proof: We prove by induction that the obfuscation of an edge from $G_1$ is an edge of $G_2$.

Let edge $(l_i, l_{i+1}) \in G_1$, where $l_i$ and $l_{i+1}$ are labels of two nodes of $G_1$. Applying Eq. 8.2 gives:

$$b_i(l_i, c_i, m_i) = l_{i+1}$$

(8.5)

Applying Eq. 8.3 for $(l_i, l_{i+1})$, we get:

$$b_i(E(l_i), c_i, m_i) = E(l_{i+1})$$

(8.6)

Based on Definition 8.7 we get:

$$\Phi(l_i, l_{i+1}) = (E(l_i), E(l_{i+1}))$$

(8.7)

Eq.8.7 states that the encrypting of edge $(l_i, l_{i+1})$ is $(E(l_i), E(l_{i+1}))$.

Combining Eq. 8.7 and Eq. 8.6, we get:

$$\Phi(l_i, l_{i+1}) = (E(l_i), b_i(E(l_i), c_i, m_i))$$

(8.8)

Since $E(l_i)$ is a vertices of Graph $G_2$, then, $G_2$ must include a statement with label $b_i(E(l_i), c_i, m_i)$. This gives:

$$(E(l_i), b_i(E(l_i), c_i, m_i)) \in A_2$$

(8.9)

Combining Eq.8.8 and Eq. 8.9 we get:

$$\Phi(l_i, l_{i+1}) \in A_2$$

(8.10)

Q.E.D.
Now we generalize Lemma 8.1 to paths of a graph

**Lemma 8.2**

Let $P_1$ and $P_2$ be the set of paths of CFGs $G_1$ and $G_2$, respectively. For any path $p$ of $P_1$ the obfuscation of $p$, $\Psi(p)$, is a path of $P_2$.

Formally:

$$\forall p \in P_1 \Rightarrow \Psi(p) \in P_2$$

**Proof:** We prove by induction

Let $p = (l_1 l_2 l_3 ... l_{m-1} l_m)$ be the label of a (any) path of $P_1$. By Definition 8.3, we can represent this path as a set of tuples:

$$p = (l_1, l_2)(l_2, l_3) ... (l_{m-1}, l_m) \quad (8.10)$$

Applying Definition 8.8 we get:

$$\Psi(l_1 l_2 l_3 ... l_{m-1} l_m) = \Phi(l_1, l_2) \Phi(l_2, l_3) ... \Phi(l_{m-1}, l_m) \quad (8.11)$$

Applying Definition 8.7 we get:

$$\Psi(l_1 l_2 l_3 ... l_{m-1} l_m)$$

$$= (E(l_1), E(l_2))(E(l_2), E(l_3)) ... (E(l_{m-1}), E(l_m)) \quad (8.12)$$

Applying Lemma 8.1 on eq. 8.11, we get:

$$\Phi(l_i, l_{i+1}) \in A_2$$

... and

$$\Phi(l_m, l_{m+1}) \in A_2$$

... and

$$\Phi(l_{m-1}, l_m) \in A_2 \quad (8.13)$$
Note that based on Eq. 8.12, the edges of the obfuscated path are connected.

Applying Definition 8.4 to Eq. 8.13 gives

\[ \Psi(p) \in P_2 \]  

(8.14)

Eq. 8.14 states that an arbitrary obfuscated path from the graph \( G_i \) is a path of graph \( O(G_i) \). Combining Eq. 8.10 and Eq. 8.13 gives:

\[ \forall p \in P_1 \Rightarrow \Psi(p) \in P_2 \]  

(8.15)

Eq. 8.15 states that the obfuscation, \( \Psi(p) \), of any path \( p \) member of \( P_1 \), is a path member of \( P_2 \).

Q.E.D.

In the following we investigate the use of privacy homomorphisms for obfuscating programs. Recall that the privacy homomorphism that we use in this chapter is defined by the branching table.

**Theorem 8.1**

*Assume that an encrypted branching table is used for computing \( \Gamma(\Pi) \) in Step 5. Then, there exists a privacy homomorphism that can be used to obfuscate program \( \Pi \).*

**Proof:** Now we prove there is a privacy homomorphism that can obfuscate programs.

Let \( \Pi_i \) be the program to obfuscate, \( \Pi_2 \) be the obfuscated program (obfuscated using the privacy homomorphism), \( G_i \) be CFG of \( \Pi_i \), \( P_i \) be the set of paths of \( G_i \), \( G_2 \) be CFG of \( \Pi_2 \), \( P_2 \) be the set of paths of \( G_2 \).

Lemma 8.2 states that:

\[ \forall p \in P_1 \Rightarrow \Psi(p) \in P_2 \]  

(8.16)
Applying Definition 8.5 on Eq. 8.16 gives:

\[ G_2 \approx > O(G_1) \quad (8.17) \]

Note that \( p \) is specified in this proof as a sequence of labels of statements. But in Definition 8.5 we use a path as a sequence of sentences.

Applying Definition 8.6 gives

\[ \Pi_2 = \Gamma(\Pi_1) \quad (8.18) \]

Q.E.D.

We conclude that if the CFG is obfuscated using the approach proposed in Section 8.3 the obfuscated program \( O(\Pi) \) preserves the functionality of the original program.

Note that computing the encrypted branching table (in this Case 1) does not require using SAS VM (see Chapter 7 for details on SAS VM); since we do not require having plaintext output of the branching functions.

Eq. 8.18 states that the functionality of the original program is preserved by the obfuscated program, which is the first property of obfuscation (see the definition of program obfuscation in Section 8.1).

We do not discuss in this chapter if the proposed solution is resilient to reverse engineering attacks that attempt to extract the original program from the obfuscated program. A solution for selecting appropriate dummy statements is required to have a full solution for obfuscating CFGs using privacy homomorphisms. Once we have a full solution we can investigate its resilience to reverse engineering.

We are working now on a solution using so called “prime programs.” A prime program is a program (sequential or parallel) that cannot be decomposed into
independent sequential programs, where two (or more) programs are independent if they do not have any input/output relationships.

**Case 2. Computing with Decrypted Labels of the Statements**

In this case, we decrypt the statement labels in order to use them for computing $\Gamma(\Pi)$. Assume that we have SAS VM that computes the encrypted branching table. Eq. 8.19 below formulates the computing.

**Corollary 8.1**

Assume that labels of statements from the branching table are decrypted in Step 5 when computing $O(\Pi)$. Then, there is no universal privacy-homomorphic algorithm for decryption of branching functions.

**Proof:** Combining Eq. 8.4 and Eq. 7.1 ($D$ is the decryption function implemented by SAS VM) gives:

$$D(b_i(E(l_j), c_i, m_i) = D(E(l_j))$$  \hspace{1cm} (8.19)

Since branching function $b_i$ is sequential, then Theorem 7.1 shows that there is no universal privacy-homomorphic algorithm for decryption of $b_i$.

Q.E.D.

Corollary 8.1 tells us that we may encrypt the statement labels for a program $P$ to form an encrypted program (using our solution). However, it is not possible to have an encrypted virtual machine (SAS VM) that can evaluate the encrypted program.
8.5. Protecting Confidentiality Using the Obfuscated CFGs and Generic Encrypted Functions

This section uses secure computing with encrypted functions (CEF), which was discussed in Subsection 3.2.2.

We investigate here the use of secure computing with encrypted functions (CEF): (i) to encrypt the branching functions of a program \( P \), and (ii) to compute the branching table for the CFG of \( P \).

The CFG is represented as a branching table. We can encrypt the branching table using generic encrypted functions, thus protecting confidentiality of obfuscated CFGs. For example the function composition method (see Subsection 3.3.8), which is an implementation of generic encrypted functions can be used to encrypt the branching table.

**Corollary 8.2**

Assume that labels of statements from the branching table are decrypted in Step 5 when computing \( O(II) \). Then, there is no universal encrypted sequential function for decryption of branching functions.

**Proof:** Let \( II \) be a program to be obfuscated, \( G \) be the flow graph of \( II \), \( O \) be a CFG obfuscator, \( E \) be the encryption function to be used to encrypt the branching table, and \( D \) be the decryption function. Let \( l_i \) and \( l_j \) be two statement labels such that:

\[
b_i(l_i, c_i, m_i) = l_j
\]

Applying Eq. 7.16 on Eq. 8.2 gives:

\[
D(E(b_i)(l_i, c_i, m_i)) = b_i(l_i, c_i, m_i) \quad (8.18)
\]
Since branching function $b_i$ is sequential, then Theorem 7.2 shows that there is no universal encrypted decryption function for $b_i$.

Q.E.D.

Corollary 8.2 tells us that we may encrypt branching functions of a program $P$ to form an encrypted program. However, it is not possible to have an encrypted virtual machine (SAS VM) that can evaluate this encrypted program.

8.6. Discussion of the Results

Theorem 8.1 states that we can obfuscate programs using privacy homomorphisms. We show that it is possible to construct an obfuscated program from an original program using a privacy homomorphism.

The results of this chapter show that a privacy homomorphism can be used to obfuscate CFGs of programs, producing obfuscated programs that preserve the functionality of the original programs.

We believe that the reliability of obfuscation using privacy homomorphisms is based on the difficulty of solving the problem of graph isomorphism. However, such obfuscation can become unreliable if discovering the relationship between the initial nodes of the CFG for the original program and the initial nodes of the CFG for the obfuscated program becomes easy.

59 Isomorphism $f$ of graphs $G$ and $H$ is a bijection between the vertex sets of $G$ and $H$ such that any two vertices $u$ and $v$ of $G$ are adjacent in $G$ if and only if $f(u)$ and $f(v)$ are adjacent in $H$ [Wiki2010g]. Bijection $g$ is a function from set $X$ to set $Y$ such that for each $y$ in $Y$ there is exactly one $x$ in $X$ such that $g(x) = y$. 

191
8.7. Conclusions

This chapter presents our solution for obfuscating programs by obfuscating their control flow graphs (CFGs). CFGs are obfuscated using secure computing approaches (CED, CEF, and CEDEF). To the best of our knowledge, we are the first to propose the use of secure computing for program obfuscation.

We believe that it is difficult to produce a sequential program that is an obfuscation of another sequential program such that the size of the obfuscated program is polynomial in the size of the original program.
9.1. Introduction

This chapter discusses a sample application using the active bundle scheme: identity management in cloud computing. Its goal is protecting private identity information used to identify an entity. In this introduction, we discuss identity management principles and identity management in cloud computing. We then present contributions and organization of the chapter.

This chapter is extracted from [ABRS10] and [RBBL10].

9.1.1. Identity Management

An identity is a set of unique characteristics of an entity: an individual, a subject, or an object. An identity used for identification purposes is called an identifier [JoPo05].

Entity identifiers are used, among others, for authentication of entities to service providers (SPs). Identifiers provide assurance (although not always a 100% guarantee) to an SP about the entity’s identity, which helps the SP to decide whether to permit a given entity to use an SP’s service or not.

Entities may have multiple digital identities. An Identity Management (IDM) System supports management of multiple digital identities. It also decides which personally
identifiable information (PII) item or items (PII is a set of data items) it is best to disclose to obtain a particular service. IDM performs the following tasks [wiki10f]:

1) **Establishing identities**: Associates a PII item with entities. For example, it associates a social security number or a name with an individual or an IP address with a host. An entity may have one or many identities. For example, a person, who is a citizen of two countries is assigned two passport numbers; each passport number identifies the person with the country that issued the number.

2) **Describing identities**: Assigns attributes (i.e., identifier) identifying an entity. Examples of such attributes are address, and age.

3) **Recording use of identity data**: Logs identity activity in a system and/or provides access to the logs. (Examples of identity activities are disclosure of a PII to an SP).

4) **Destroying identity**: Assigns expiration dates to PIIs. All PII items become unusable after PII expiration date.

![Figure 9.1. Authentication using third party identity management](image)

Fig. 9.1 shows an example of authentication that uses PII. A user wants to use a service, for which she needs to be authenticated by a SP, but does not want to disclose all her PII items. She has to disclose a sufficient subset of PII items to uniquely identify
herself to the SP. The main problem is to decide which PII items to disclose, and how to disclose them in a confidential way.

Different parties use IDM and collaborate to identify and authenticate an entity. These parties are (cf. [CaJo06]):

1) *Identity provider (IdP).* It issues digital identities. For instance, credit card providers issue identities (credit card numbers) enabling payments, governments issue identities (including SSNs) to citizens.

2) *Service provider (SP).* It provides services to entities that have required identities. For instance, a user needs to show her SSN (one of her PII items) to a SP providing tax filing online.

3) *Entity about which claims are made.* A claim could be, for example, a name, a date of birth, or SSN. An entity proclaims a PII item about herself. A claim is a PII item.

4) *Identity verifier.* It receives requests from a SP for verifying a claim from a specific entity. It verifies correctness of the claim, and decides whether the claim is correct or not.

IDM uses one of the following three categories of identifiers: (1) information that both an entity and SP know, such as passwords; (2) information that an entity knows and a SP can get approved by IdP, such as the entity’s SSN; and (3) other information about the entity, such as fingerprints.

60 “Identify” implies losing one’s anonymity. One can be authenticated by providing a subset of PII, and thus avoid being identified.
9.1.2. Identity Management in Cloud Computing

In the traditional application-centric [Gop09] IDM model, each application keeps track of entities that use it. In cloud computing, entities may have multiple accounts (and identities for using them) associated with different SPs. Entities may also use multiple services offered by the same SP (e.g., Gmail and Google Docs can be offered by the same SP: Google).

A cloud user has to provide a subset of his PII items, which authenticates him, while requesting any service from the cloud. Any entity using many cloud services leaves behind a trail of disclosed PII items that can be collected and exploited to uniquely identify, locate, or track the entity. The trail of PII items—if not properly protected—may be exploited and abused. Sharing different PII items of a given entity across services can lead to identifying the entity [Gop09]. The main issue is how to protect PII from being disclosed to unauthorized parties. (More precisely, we want to prevent situations that a subset of disclosed PII items identifies the entity described by this PII.) If we fail, PII disclosure might result in serious crimes against privacy, including identity theft [Gel09].

In cloud computing, the burden of assuring privacy of data rests on the owner of these data. To ease the burden, the owner could be given technical controls supporting this challenging task.

We believe that cloud computing requires an entity-centric IDM model, in which every entity’s request for any service is bundled with the entity’s identity [LAW05]. We propose here an entity-centric IDM that: (1) creates and manages digital identities of entities; (2) authenticates entities in a way that does not reveal their actual identities or relationships between the entities (or their real identities) and vendors, service providers, etc; and (3)
protects an entity’s PII from unauthorized accesses and disclosures. Fig. 9.2 compares the basic structure of application-centric and entity-centric IDMs. The advantage of an entity-centric IDM is that a disclosure of PII items is no longer arbitrary or at the will of SP but instead is controlled by the PII owner. This chapter proposes an approach for designing an entity-centric IDM model.

9.1.3. Contribution and Chapter Organization

We propose an approach for IDM in cloud computing that: (1) does not use TTPs (trusted third parties); and (2) can be used on untrusted or unknown hosts. The approach is based on two basic features. First, it uses the active bundle scheme—with a bundle including PII, privacy policy and a VM (a virtual machine) that enforces the bundle’s policy and implements a set of protection mechanisms (including integrity checks, evaporation, apoptosis, and decoy) for the bundle. Second, it relies on anonymity-preserving authentication to mediate interactions between an entity and cloud services that follow the entity’s privacy policies.

![Figure 9.2. Application-centric IDM vs. entity-centric IDM](image-url)
This chapter is organized as follows. Section 9.2 discusses related work. Section 9.3 presents our approach proposed for protecting PII in cloud computing. Section 9.4 concludes the chapter.

9.2. Related Work

The best known projects and solutions for IDM, presented by us in some detail elsewhere [ABRS10] are:

1) **PRIME (Privacy and Identity Management for Europe):** PRIME [Fisc10] provides privacy-preserving authentication using a TTP-based IdP (identity provider).

2) **Windows CardSpace:** Windows CardSpace [AlMi09] treats every digital identity as a security token, which consists of a set of claims (such as a username, a full name, an address, SSN, etc.). Tokens prove to a SP that the claims that they use belong to the user presenting the claims.

3) **Open ID:** Open ID [Open10] is a decentralized authentication protocol that helps cloud users in managing their multiple digital identities with a greater control over sharing their PII items. A user has to remember only one username and password—an OpenID—to log onto all websites accepting this OpenID.

9.3. The Proposed Approach for Protecting PII

This section describes the approach proposed for an entity-centric IDM model using the active bundle scheme and the anonymity-preserving authentication method.
9.3.1. Characteristics of the Existing Approaches

The solutions that we reviewed have two important characteristics:

1) Use of TTPs. There are two major issues for adopting such an approach for cloud computing. First, a TTP (which could be a cloud service located at the cloud provider’s host) and a certain SP may be the same (the TTP and SP are the same entity rather than different entities).

   In such a case, the TTP is no longer an independent trusted entity\(^{61}\) (not independent from the SP). Second, it is a centralized approach, because it uses one entity (i.e. the TTP) when authenticating to all SPs. Therefore, if the TTP is compromised, PII items for its users are compromised too.

2) No support for untrusted hosts. A client application that holds PII must be executed on a trusted host.

9.3.2. Selected Research Problems

We investigate the following selected problems in this chapter:

1) Authentication without disclosing (unencrypted) data: When a user sends PII items for authentication for a service by a SP, the user may encrypt these items. However, before any PII items are used by the SP, they must be decrypted. Plaintext PII items become an easy target for attacks. This is of a particular concern if the SP decides to store decrypted PII items.

2) Using services on untrusted hosts (not owned by the user of authentication or input data given to services): The earlier IDM solutions known to us require that the user

\(^{61}\) Independent trusted entity is a neutral entity, i.e., does not side with one of the parties that trust it.
wishing to use the IDM system or service be on a trusted host. They do not allow using IDM on untrusted hosts, such as public hosts. With the advances in cloud computing—in which data may reside anywhere in the cloud, also on a public host—this issue needs to be addressed.

3) Minimizing risks of PII disclosure during communication between users and service providers (protecting from side channel and correlation attacks): PII items must be protected from unauthorized disclosures. This becomes even more important in cloud computing, where sensitive data may be stored by a SP, and then transmitted to another SP (e.g., a subcontractor), to use the other SP’s service.

9.3.3. The Proposed Entity-centric IDM Approach

Our approach to entity-centric IDM, which we name IDM Wallet, uses the active bundle scheme to protect PII from untrusted hosts. We use anonymity-preserving authentication of an entity without disclosing its PII. Fig. 9.3 shows the structure of IDM Wallet, and illustrates anonymity-preserving authentication.

Using anonymity-preserving authentication, it is possible to prove a claim or an assertion (i.e., to authenticate the entity that provides its claim or assertion) without actually disclosing any entity identity.

As an example, suppose that a user buys books online from Amazon. She needs to provide her address to receive the books by mail. This is one situation where multiple PII items must be protected from unauthorized disclosures. This becomes even more important in cloud computing, where sensitive data may be stored by a SP, and then transmitted to another SP (e.g., a subcontractor), to use the other SP’s service.

Anonymity-preserving authentication relies on using only a subset of PII items for any authentication, a subset that does not identify the PII owner (the subset is anonymity-preserving as well). We describe this scheme in Subsection 9.3.5.
parties are involved in the same transaction and need different information from the user. The shipping company needs to know the user’s address. Amazon needs no address but wants to be sure that she gives a legitimate address to the shipping company.

When IDM Wallet is used in this example, it starts with anonymity-preserving identification. Then, IDM Wallet creates an active bundle (AB) that includes PII (in this case, PII consists of one item only: the user’s address) that needs to be disclosed for authentication. The AB includes PII, metadata (with access control policies), and a VM. The AB is given to the SP that forwards the AB (without looking at its contents) to the shipping company. Even if the SP is an untrusted host, the address is protected. This illustrates that IDM Wallet gives us protection even for SPs running on untrusted hosts (actually, this protection is the result of disseminating only PII within ABs encapsulating it).

Figure 9.3. Structure of IDM Wallet and anonymity-preserving service interaction.

9.3.4. Description of IDM Wallet

IDM Wallet uses active bundles, which hold users’ PIIIs and manage their disclosures.
The structure of IDM Wallet, shown in Fig.9.3, includes the following components:

1) **Identity data:** PII used for authentication, getting service, and using service (e.g., SSN, date of birth). PII is encrypted and enclosed inside the IDM Wallet (directly, within an active bundle, which is stored by IDM Wallet).

2) **Disclosure policy:** This is a set of rules for choosing specific PII items from the set of all PII items in IDM Wallet. For instance, if certain PII items are used for service $S$, then the same PII items must be disclosed for the same service $S$ every time it is used. Disclosing other PII items to that service increases the risks of the full PII disclosure to it.

3) **Disclosure history:** History of disclosures of PII items to services is logged. It is critical data used for the selection of PII items to be disclosed for authentication. It is also used for auditing purposes.

4) **Negotiation policy:** It is used for anonymity-preserving authentication.

5) **VM code:** It is the active bundle’s VM code for protecting PII items on untrusted hosts. It enforces disclosure policies.

### 9.3.5. Description of Anonymity-preserving Authentication

This subsection describes Fiat and Shamir’s identification and signature scheme [FiSh86]. Then, it discusses using their scheme for anonymity-preserving authentication used in IDM Wallet.
Protocol 1 (issuing PII by an IdP to an entity)

P1.1) IdP checks the physical identity of an entity and prepares a string $I$, which contains all relevant information about the entity (which also includes the validity conditions for PII (expiration date, limitations on validity, etc).

P1.2) IdP then performs the following steps:

   a) Compute values $v_j = f(I, j)$ for information at indices $j$

   b) Pick $k$ distinct values of $j$ for which $v_j$ is a quadratic residue (mod $n$) and compute the smallest square root of $s_j$ of $v_j^{-1}$.

   c) Issue PII, which contains $I$, the $k$ $s_j$ values and the selected $k$ indices.

   (To simplify notation we assume that the first $k$ indices $j = 1, 2, \ldots, k$ are used.)

Figure 9.4. Protocol 1 (issuing PII by an IdP to an entity) from the Fiat-Shamir identification scheme

9.3.5.1. Description of Fiat-Shamir Identification Scheme

The goal of Fiat and Shamir identification scheme is to allow SP to verify PII of an entity. The scheme prevents SP from using PII of the customer to identify her [FiSh86]. The scheme includes two protocols: issuing PII to an entity, and verifying PII of an entity. Before issuing PII, IdP (identity provider) chooses an integer $n$ and a pseudo random function $f$, where $f$ associates arbitrary strings with integers from the range $[0,n)$. (For

---

$63$ Recall that $I$ is a list of attributes (credentials) where each element of the list (a credential) has an index $j$ (its position in the list).

$64$ $q$ is quadratic residue modulo $n$ if there exist $x$ such that $x^2 \equiv q \mod n$. E.g., $2$ is a quadratic residue modulo $7$ because $3^2 \equiv 2 \mod 7$
example, \( f \) associates “abc” to “1”. The selected integer \( n \) must be the product of two secret prime numbers \( p \) and \( q \).

Protocols 1 and 2 are shown in Fig. 9.4 and 9.5, respectively.

Protocol 2 (verifying the identity of an entity)

Assume that the SP has the universal modulus\(^65\) \( n \) and universal function \( f \). The goal of this step is that the entity (Party A) proves to the SP (Party B) that it is the owner of PII \( I \).

The steps of the protocol are:

P2.1) A sends \( I \) to B

P2.2) B generates \( v_{j} = f(I, j) \) for \( j = 1, \ldots, k \)

P2.3) \textbf{for} \( i = 1, \ldots, t \) \textbf{do}:

a) A picks a random \( r_{i} \in [0, n) \)

b) A sends to B: \( x^{i} = r_{i}^{2} \pmod{n} \)

c) B sends to A: a random binary vector \((e_{i1}, \ldots, e_{ik})\)

d) A sends to B: \( y_{i} = r_{i} \prod_{e_{ij} = 1} s_{j} \pmod{n} \)

e) B checks that \( x_{i} = y_{i}^{2} \prod_{e_{ij} = 1} v_{j} \pmod{n} \)

Figure 9.5. Protocol 2 (verifying the identity of an entity) from the Fiat-Shamir identification scheme

In the Fiat and Shamir scheme, an entity (Party A) gives information \( I \) to the SP (Party B). B verifies that IdP issued \( I \) for A. Since A does not know how to compute \( s_{j} \) from \( v_{j} \)

\(^65\) The \textit{universal modulus} is an integer known to all parties (same applies for function \( f \)).
because it does not know the primes that compose $n$. If all $t$ checks are successful, then B knows at the end of the protocol that $I$ indeed authenticated A.

9.3.5.2. Outline of the Proposed Anonymity-preserving Authentication Scheme

We adapt here the Fiat and Shamir’s identification scheme [FiSh86] in such a way that Party A does not give information $I$ to Party B (in contrast to Step 1 of Protocol 2). The protocol verifies anonymously that the provided PII items authenticate their owner.

Our proposed new anonymity-preserving authentication scheme allows any SP to authenticate entities without knowing their identities (e.g., their complete PII data). The scheme includes two protocols. We use Protocol 1 of Fiat and Shamir as is, but modify Protocol 2 as shown in Fig. 9.6.

Modified Protocol 2 (verifying the identity of an entity)

MP2.1) IdP sends to B: $v_j = f(I, j)$ for $j = 1, \ldots, n$.

MP2.2) for $i = 1, \ldots, t$ do

a) A picks a random $r_i \in [0, n)$

b) A sends to B: $x_i = r_i^2 \pmod{n}$

c) B sends to A: a random binary vector $(e_{i1}, \ldots, e_{ik})$

d) A sends to B: $y_i = r_i \prod_{e_{ij} = 1} s_j \pmod{n}$

e) B checks that $x_i = y_i^2 \prod_{e_{ij} = 1} v_j \pmod{n}$

Figure 9.6. Modified Protocol 2 (issuing PII by an IdP to an entity)
Note that Modified Protocol 2 does not include Steps P2.1 and P2.2 of Protocol 2. In our proposed scheme, A does not share his PII with SP. We replace Steps P2.1 and P2.2 with a new step MP2.1. In Step MP2.1, IdP provides B with the set of strings \( \{v_j\} \) representing the entity’s PII.

In the Modified Protocol 2, the role of A is to match one of its PII with one of the legitimate values held by the SP (acquired by the SP from IdP in Step MP2.1).

**9.3.6. Simulating the Use of the Proposed Approach for Entity-centric IDM**

The following is a scenario simulating the use of the proposed approach:

1) An entity requests a service from the SP (e.g., a website).

2) In response to the service request, the SP informs the entity that in order to gain access to the service the entity must provide a set of credentials for authentication.

3) IDM Wallet evaluates the owner preferences which are described as a privacy policy included in the Wallet and decides on the set of credentials that it should disclose.

4) IDM wallet interacts with the SP based on Modified Protocol 2 such that it convince the SP that it holds the credentials without disclosing their values. The SP authenticates the owner of the IDM Wallet if the verification of the credentials is successful.

5) If further information is required by the SP on behalf of a third party (e.g., an individual orders a book from Amazon. Amazon requests the individual’s
address which is to be used by the shipping company to ship the book), IDM Wallet creates an AB that includes the information and forwards it to the SP. The SP forwards the information to the third party.

9.3.7. **Characteristics and Advantages of the Proposed IDM**

There are two salient features of the proposed entity-centric IDM Wallet solution. First, IDM Wallet is able to use PII on untrusted hosts. It includes a self-integrity check to find out if PII was tampered with. If integrity is compromised, PII will self-destroy partially (by evaporation) or completely (by apoptosis) to protect endangered PII items from falling into the wrong hands (Chapter 4 gives more details about these operations performed by active bundles).

Second, IDM Wallet is independent from TTPs (does not need them to operate). This protects PII data from correlation attacks, side-channel attacks, and problems that could be caused by a third party being compromised. It increases a user’s trust level by putting the user in control of who uses and how his PII items in the process of PII dissemination, user authentication, and service use negotiation.

Advantages of the proposed entity-centric IDM Wallet solution include the following:

1) *Independent and trustworthy*. The interaction is only between a user and SP. This is for example in contrast to OPENID where a trusted third party verifies the credentials of entities requesting a SP.

2) *Gives minimum information to the SP*. The SP receives only necessary information. The fully developed IDM Wallet (versus. just outlined here) will include details on negotiation that will achieve this.
3) **Portability.** IDM Wallet can be carried on a mobile or flash drive etc. It includes all the required data and code. It could be executed on any machine.

### 9.4. Conclusions

With the immense growth in the popularity of cloud computing, privacy and security have become a critical concern for both the public and private sector. There is a strong need for an effective and efficient privacy-preserving identity management (IDM) system. In fact, IDM must be one of the core components assuring required levels of privacy and security in cloud computing.

We believe that such a system for cloud computing should be based on an entity-centric mechanism for protecting confidentiality of Personally Identifiable Information (PII). The system should: (1) be independent of any TTP; (2) be able to facilitate user authentication both across the Web and within enterprises; and (3) be able to protect users’ PII items.
10.1. Conclusions

Information sharing is facing serious challenges in securing disseminated data. The great promise of ubiquitous computing—one of the most important challenges in computing nowadays—will not be realized without information sharing. The main challenge is how to share data while protecting them from unauthorized disclosures.

This Thesis investigates confidentiality of sensitive data. Confidentiality means ensuring that information is not made available or disclosed to unauthorized entities, whether human or artificial.

We propose a solution for protecting sensitive data during their dissemination using the active bundle scheme. An active bundle packages together sensitive data, metadata, and a virtual machine (VM) specific to the bundle. Metadata contain information about data. They include, among others, privacy policies for controlling access to sensitive data and for their dissemination. The VM manages operations of its active bundle, enforcing the policies specified by metadata.

The active bundle scheme protects private or sensitive data from their disclosures to unauthorized parties and from unauthorized dissemination (even if started by an authorized party). The scheme protects private data throughout their entire lifecycle, from creation through dissemination to their partial or total destruction.
Implementing bundles’ VM is the largest challenge for building effective and efficient active bundles. We investigated different approaches to VM implementation: (1) using trusted third parties (TTPs), (2) utilizing mobile agents (3) using autonomous applications based on secure computing, and (4) using autonomous applications based on obfuscated control flow graphs.

We developed a prototype that demonstrates feasibility of developing a TTP-based implementation of the active bundle scheme. The prototype shows that the scheme protects confidentiality of sensitive data even when unauthorized entities attempt to gain access to them.

The main contributions of this Thesis are:

1) We propose a scheme for protecting sensitive data throughout their lifecycle. The main features of the scheme are: (i) enforcing privacy policies, (ii) activating protection mechanisms when data is tampered with, and (iii) recording active bundle activities for audits. Embedding VMs within active bundles allows protection of sensitive data within active bundles even from malicious hosts.

2) We answer the following two questions: (i) Are there any available solutions for protecting confidentiality of code and carried data for mobile agents providing output to visited hosts that satisfy the two critical confidentiality-related mobile agent properties: computation autonomy and data autonomy? (ii) Are any of these solutions practical?

The answer to the first question is “Yes,” but the response to the second question is “No.”
3) We developed a TTP-based prototype of the active bundle scheme. The prototype demonstrates practicality of the scheme by being able to protect confidentiality of disseminated sensitive data when unauthorized entities attempt to access these data.

4) We answer negatively the following two questions: (i) Is there a universal privacy-homomorphic decryption function? (ii) Is there a secure autonomous sequential virtual machine for an encrypted decryption function?

5) We pioneer—to the best of our knowledge—the use of secure computing for program obfuscation.

6) We present a sample application of active bundles for identity management.

We believe that these contributions justify the following thesis of this Thesis:

Data can protect themselves from unauthorized accesses by malicious hosts.

This is possible because active bundles make data inseparable from metadata and VMs. VMs make data active, able to protect themselves from unauthorized accesses by malicious hosts.

10.2. Directions for Future Work

The first subsection discusses future work on improving ABTTP, the second subsection—future work on improving the active bundle scheme, and the third subsection—future work on improving security of the active bundle scheme.

10.2.1. Future Work on Improving ABTTP

Our future work on improving capabilities of ABTTP includes the following:
1) ABTTP does not implement evaporation and decoy mechanisms. We plan to develop mechanisms for evaporation and decoy.

2) We plan to develop integrity checking mechanisms for verification of VM originality (i.e., is the code as created by the AB Creator’s owner?) This will assure that an active bundle does not run if its code was tampered with.

3) ABTTP simulates but does not include enforcement of privacy policies. We plan to extend ABTTP with a formal language for specifying privacy policies, and with mechanisms to enforce the policies.

4) We plan to measure performance of ABTTP in different deployments setups (as defined in Section 6.4). We will investigate how changing the distribution of ABTTP components among hosts (including changing the number of hosts) affects performance of the prototype. We will also measure how the number of created active bundles affects performance of ABTTP components, and ABTTP as the whole.

5) ABTTP records audit information about active bundle activities. However, ABTTP does not include tools that owners can use to analyze recorded audit information. We plan to develop such tools (they will be a part of AB Client).

6) ABTTP simulates a trust evaluation agent. We plan to develop mechanisms for trust evaluation of destination hosts.

7) We plan to adapt ABTTP and develop experiments for protecting confidentiality of data in sample applications, such as the smart grid and pervasive health monitoring.
10.2.2. Future Work on Improving the Active Bundle Scheme

The main task for our future work in the area of improving the active bundle scheme is developing an obfuscation approach for obfuscating the bundles’ VM code. We are investigating two approaches to obfuscation. For the first approach, based on may-aliases, we plan to propose obfuscation that uses dereferencing, function pointers, and recursive functions; and to investigate the efficiency of the approach. This choice is motivated by the fact that alias analysis techniques (including the ones that use Binary Decision Diagram, BDD) provide approximation of memory allocation when dereferencing, function pointers, and recursive functions are used.

The second approach to obfuscation is based on using non-decomposable concurrent programming techniques. We propose searching for efficient algorithms able to add “foolproof” dependencies among two (or more) component programs (the original and padding programs) via branching statements.

10.2.3. Future Work on Improving Security of the Active Bundle Scheme

Current approaches for protecting sensitive data assume that the host receiving them is trusted, and will do all that is expected of it to enforce the privacy policies associated with data. In contrast, the active bundle scheme relaxes this assumption (under the conditions of our threat model).

In our future work, we plan to analyze security of the active bundle scheme. Our primary interests will be evaluation of data leaks from active bundles, and understanding attacks on bundles. For example, active bundles are not resistant to some attacks by
malware. An execution of a bundle’s VM on a host can be altered by malware so that it can gain unauthorized access to bundle’s data.

As another form of attack, a bundle’s VM might be denied execution at the host it visits. In this case, its data are not disclosed to unauthorized entities but legitimate entities are denied access to these data. We plan to study different attacks and provide solutions for countering these attacks.
BIBLIOGRAPHY


[OHRP2009] Division of Compliance Oversight, Office for Human Research Protections (OHRP)


