Flexible Information-Flow Control

DANIEL SCHOEPE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND GÖTEBOURG UNIVERSITY

Göteborg, Sweden 2018
As more and more sensitive data is handled by software, its trustworthiness becomes an increasingly important concern. This thesis presents work on ensuring that information processed by computing systems is not disclosed to third parties without the user’s permission; i.e. to prevent unwanted flows of information. While this problem is widely studied, proposed rigorous information-flow control approaches that enforce strong security properties like noninterference have yet to see widespread practical use. Conversely, lightweight techniques such as taint tracking are more prevalent in practice, but lack formal underpinnings, making it unclear what guarantees they provide.

This thesis aims to shrink the gap between heavyweight information-flow control approaches that have been proven sound and lightweight practical techniques without formal guarantees such as taint tracking. This thesis attempts to reconcile these areas by (a) providing formal foundations to taint tracking approaches, (b) extending information-flow control techniques to more realistic languages and settings, and (c) exploring security policies and mechanisms that fall in between information-flow control and taint tracking and investigating what trade-offs they incur.
This thesis is based on work contained in the following papers, each presented in a separate chapter. Chapters 1, 2, 4, 6, and 7 are published at peer-reviewed conferences while the content of the other papers is currently under submission.


**Chapter 5** Information-Flow Control for Database-Backed Applications. Marco Guarnieri, Daniel Schoepe, Musard Balliu, David Basin, and Andrei Sabelfeld. Under submission.


The past four and a half years have been a unique experience: often exciting, sometimes frustrating, but always memorable. Spending so much time on a single topic can be challenging and I owe a debt of gratitude to many people for helping me to successfully finish my PhD.

Firstly, I want to thank my advisor Andrei for years of great supervision and advice and countless interesting discussions not just about research, but also about food, drinks, politics, and the rest of life. He also provided me with lots of exciting opportunities to work with other researchers and helped me organize research visits and internships. I also owe a debt of gratitude to Musard, Dave, and Wolfgang, for the many fruitful conversations and lots of helpful advice during my PhD.

Next, I want to thank the many great people that make (or made) coming to work each day feel not just like an obligation, but actually fun: Alejandro, Alexander, Aljoscha, Daniel, Elena, Elisabet, Evgeny, Herbert, Hiva, Inari, Iulia, Jeff, Katja, Marco, Mauricio, Max, Michal, Pablo, Per, Raul, Sandro, Simon, Solrún, Steven, Thomas, Víctor, and many others I doubtlessly forgot to mention here.

Being caught up in research and teaching all day makes it all too easy to forget that there’s also a life outside of work and I’m grateful to have found so many friends over the years, so I want to thank Noémi, Eike, Nicole, Baldur, Елена, Brian, Irene, Fotini, Marianna, Alla, Ari, Edit, Maria, Rhys, among many others, for putting up with me over all these years. Lastly, I’m grateful to my parents for supporting me throughout this endeavor and many other challenges throughout my life.

At the end of this long road, I’m happy to have taken on this challenge, but also to finally bring it to a conclusion in the form of this thesis.
# Introduction

## 0.1 Language-Based Security

## 0.2 Information-Flow Control

## 0.3 Taint Tracking

## 0.4 Challenges

## 0.5 Contributions

### 0.5.1 Explicit Secrecy: A Policy for Taint Tracking

### 0.5.2 Let’s Face It: Faceted Values for Taint Tracking

### 0.5.3 VERONICA: Verified Concurrent Information-Flow Security Unleashed

### 0.5.4 JSLINQ: Building Secure Applications across Tiers

### 0.5.5 Information-Flow Control for Database-Backed Applications

### 0.5.6 Understanding and Enforcing Opacity

### 0.5.7 We Are Family: Relating Information-Flow Trackers

### 0.5.8 An Empirical Study of Information Flows in Real-World JavaScript

# Explicit Secrecy: A Policy for Taint Tracking

## 1.1 Introduction

## 1.2 Specifying Explicit Flows

### 1.2.1 Weak Secrecy

### 1.2.2 Explicit Secrecy

### 1.2.3 Declassification

### 1.2.4 Instantiating Explicit Secrecy

### 1.2.5 Explicit Secrecy in the Big Picture

## 1.3 Enforcement

### 1.3.1 Dynamic Tainting for Imperative Code

### 1.3.2 Dynamic Tainting for Machine Code

### 1.3.3 Static Analysis for Taint Tracking

## 1.4 Related Work

## 1.5 Conclusion

# Let’s Face It: Faceted Values for Taint Tracking

## 2.1 Introduction

## 2.2 Faceted Values for Taint Tracking

### 2.2.1 Language with Faceted Values

### 2.2.2 Explicit Secrecy

### 2.2.3 Attack Detection

### 2.2.4 Inlining Faceted Values through Static Program Transformation

## 2.3 General Framework

## 2.4 Implementation

## 2.5 Benchmarks

## 2.6 Related Work

## 2.7 Conclusion
## CONTENTS

```
Appendix 2.A  Proofs .............................................. 72
  2.A.1  Theorems in Section 2.3 .................................. 72
Appendix 2.B  DroidBench Evaluation Results ...................... 76
  2.B.1  Jimple .................................................. 76
  2.B.2  DroidBench Results .................................... 77

3  VERONICA: Verified Concurrent Information-Flow Security Unleashed 79
  3.1  Introduction ............................................... 80
  3.2  An Overview of VERONICA ................................. 81
    3.2.1  Decoupling Functional Correctness ..................... 81
    3.2.2  Compositional Enforcement ............................. 84
    3.2.3  Proving a Concurrent Program Secure .................. 85
  3.3  Security Definition ..................................... 88
    3.3.1  Semantic Model ....................................... 88
    3.3.2  System Security Property and Threat Model ........... 90
    3.3.3  Occlusion ........................................... 93
    3.3.4  Encoding Delimited Release Policies .................. 94
  3.4  Annotated Programs in VERONICA .......................... 97
  3.5  The VERONICA Logic ..................................... 98
    3.5.1  Precise Reasoning with Annotations .................... 98
    3.5.2  Secret-Dependent Branching ............................ 100
    3.5.3  Soundness ........................................... 100
  3.6  Further Examples ....................................... 101
    3.6.1  The Example of Figure 3.1 ............................. 101
    3.6.2  Confirmed Declassification ............................ 101
    3.6.3  Running Average ...................................... 103
  3.7  Related Work .......................................... 104
  3.8  Conclusion ............................................. 105
Appendix 3.A  Ancillary Definitions ................................. 105

4  JSLINQ: Building Secure Applications across Tiers 109
  4.1  Introduction ............................................. 110
  4.2  Framework ................................................ 113
    4.2.1  Language ............................................. 113
    4.2.2  Operational Semantics ................................ 116
    4.2.3  Security Condition ................................... 117
    4.2.4  Security Type System .................................. 119
    4.2.5  Soundness ............................................ 124
  4.3  JSLINQ .................................................. 124
  4.4  Case Studies ............................................ 127
    4.4.1  Library Policy ........................................ 127
    4.4.2  Scenario Discussion .................................... 128
    4.4.3  Case Study Results .................................... 130
  4.5  Related Work .......................................... 131
  4.6  Conclusion ............................................. 133
Appendix 4.A  Appendix ........................................ 133
    4.4.1  Operational Semantics ................................ 133
    4.4.2  Soundness Proof ...................................... 133

5  Information-Flow Control for Database-Backed Applications 143
  5.1  Introduction ............................................. 144
  5.2  Overview ............................................... 145
  5.3  WHILESQL ............................................... 148
    5.3.1  Notation ............................................. 148
    5.3.2  Overview ............................................. 149
    5.3.3  Local semantics ...................................... 150
    5.3.4  Global semantics ..................................... 152
  5.4  Security model ......................................... 152
    5.4.1  Preliminaries ......................................... 152
    5.4.2  Knowledge ............................................ 153
```
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.3 Security condition</td>
<td>153</td>
</tr>
<tr>
<td>5.5 Enforcement</td>
<td>154</td>
</tr>
<tr>
<td>5.5.1 Preliminaries</td>
<td>154</td>
</tr>
<tr>
<td>5.5.2 Security monitor</td>
<td>155</td>
</tr>
<tr>
<td>5.5.3 Discussion</td>
<td>159</td>
</tr>
<tr>
<td>5.6 Disclosure lattices in practice</td>
<td>160</td>
</tr>
<tr>
<td>5.6.1 Approximating disclosure lattices</td>
<td>160</td>
</tr>
<tr>
<td>5.6.2 Symbolic tuples</td>
<td>161</td>
</tr>
<tr>
<td>5.7 Implementation and case studies</td>
<td>164</td>
</tr>
<tr>
<td>5.7.1 Securing SCALA programs</td>
<td>164</td>
</tr>
<tr>
<td>5.7.2 Case studies</td>
<td>165</td>
</tr>
<tr>
<td>5.8 Related work</td>
<td>168</td>
</tr>
<tr>
<td>5.9 Conclusion</td>
<td>171</td>
</tr>
<tr>
<td>Appendix 5.A Tracking dependencies between tuples and columns</td>
<td>171</td>
</tr>
<tr>
<td>Appendix 5.B Progress-sensitivity</td>
<td>172</td>
</tr>
<tr>
<td>Appendix 5.C Relaxing NSU checks</td>
<td>172</td>
</tr>
<tr>
<td>Appendix 5.D Social Networking example</td>
<td>172</td>
</tr>
<tr>
<td>6 Understanding and Enforcing Opacity</td>
<td>177</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>178</td>
</tr>
<tr>
<td>6.2 Framework</td>
<td>181</td>
</tr>
<tr>
<td>6.2.1 Opacity</td>
<td>181</td>
</tr>
<tr>
<td>6.2.2 Noninterference</td>
<td>183</td>
</tr>
<tr>
<td>6.2.3 Main Lemma</td>
<td>183</td>
</tr>
<tr>
<td>6.2.4 Knowledge-based Characterization</td>
<td>184</td>
</tr>
<tr>
<td>6.3 Batch-job Programs</td>
<td>185</td>
</tr>
<tr>
<td>6.4 I/O Programs</td>
<td>186</td>
</tr>
<tr>
<td>6.5 Information Release vs. Information Hiding</td>
<td>187</td>
</tr>
<tr>
<td>6.6 Enforcement</td>
<td>188</td>
</tr>
<tr>
<td>6.6.1 Example Language</td>
<td>188</td>
</tr>
<tr>
<td>6.6.2 Dynamic Monitoring</td>
<td>191</td>
</tr>
<tr>
<td>6.6.3 Sampling-based Enforcement</td>
<td>194</td>
</tr>
<tr>
<td>6.7 Experiments</td>
<td>195</td>
</tr>
<tr>
<td>6.7.1 Location Privacy</td>
<td>195</td>
</tr>
<tr>
<td>6.7.2 Statistics Aggregation</td>
<td>197</td>
</tr>
<tr>
<td>6.7.3 Discussion</td>
<td>198</td>
</tr>
<tr>
<td>6.8 Related Work</td>
<td>199</td>
</tr>
<tr>
<td>6.9 Conclusions</td>
<td>201</td>
</tr>
<tr>
<td>Appendix 6.A Proofs</td>
<td>202</td>
</tr>
<tr>
<td>6.A.1 Framework</td>
<td>202</td>
</tr>
<tr>
<td>6.A.2 Enforcement</td>
<td>204</td>
</tr>
<tr>
<td>7 We Are Family: Relating Information-Flow Trackers</td>
<td>211</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>212</td>
</tr>
<tr>
<td>7.2 Security Framework</td>
<td>214</td>
</tr>
<tr>
<td>7.2.1 Language</td>
<td>214</td>
</tr>
<tr>
<td>7.2.2 Semantics</td>
<td>215</td>
</tr>
<tr>
<td>7.2.3 Defining Secrecy</td>
<td>216</td>
</tr>
<tr>
<td>7.2.4 Security Conditions</td>
<td>216</td>
</tr>
<tr>
<td>7.3 Enforcement Framework</td>
<td>217</td>
</tr>
<tr>
<td>7.4 Staged Information-Flow Control</td>
<td>221</td>
</tr>
<tr>
<td>7.5 Implementation and Evaluation</td>
<td>225</td>
</tr>
<tr>
<td>7.6 Related Work</td>
<td>226</td>
</tr>
<tr>
<td>7.7 Conclusion</td>
<td>228</td>
</tr>
<tr>
<td>Appendix 7.A Semantics</td>
<td>230</td>
</tr>
<tr>
<td>7.A.1 Operational Semantics of SIMPL</td>
<td>230</td>
</tr>
<tr>
<td>7.A.2 Instrumented Semantics for Weak and Observable secrecy</td>
<td>231</td>
</tr>
<tr>
<td>Appendix 7.B Proofs for Enforcement Mechanisms</td>
<td>231</td>
</tr>
<tr>
<td>7.B.1 Proofs for Weak Tracking</td>
<td>232</td>
</tr>
</tbody>
</table>
Developments in computer science have shaped many aspects of today’s society. In fact, it is hard to find an area of modern life that is untouched by information technology. A significant portion of our lives, our institutions, and our economy is controlled by software. On a societal level, code determines how news is disseminated, how the stock market is managed, how our healthcare systems work, and sometimes we even rely on code to safeguard democratic elections. On an individual level, software impacts what we read, how we communicate with friends, express our thoughts, find our way in a city, manage our finances, take pictures, watch movies, find out what to spend our free time on, to name just a few examples.

In the process we make a large amount of sensitive data available to software, including our location, search queries, private messages, pictures and videos, hobbies, and political views. Yet — despite the role computing technology plays — there is no practical way, not even for an expert, to ensure that all the software in use is actually trustworthy. How can one be certain that an application does not send sensitive data to a third party? A malicious application could leak confidential files on the hard drive, passwords, private messages, and credit card numbers.

Disclosing private data handled by applications can have severe consequences: Companies may lose profits if internal data is leaked. If a credit card number is stolen by a scammer, the victim will lose money. A stolen social security number can give rise to identity theft. Revealing a person’s political views, private conversations, religious beliefs, or sexual orientation might cause them to lose their job or impact their social life, or even cost them their lives in some countries. Identity and location can be particularly sensitive information; in the case of domestic abuse victims and dissidents living under an oppressive regime, their disclosure can prove quite literally life-threatening.

This problem is exacerbated as more and more adversaries attempt to compromise people’s devices and the software that runs on them; attackers range from ordinary criminals motivated by simple greed to government agencies seeking to undermine the ideals of liberty and individual rights that underpin our society. Ever since Edward Snowden’s revelations [4], we know that (even) Western governments steal their citizen’s information on a large scale [10], ostensibly
in an effort to combat terrorism, although evidence that mass surveillance is helpful in that regard has yet to surface. While cryptography is essential in defending against attackers who control the network, such protection still relies on trustworthy software on the users’ devices to perform the encryption and not leak the data in the process. Since one of the methods used for stealing data is to spread malicious or backdoored software to the users’ devices, this illustrates the need for a way of reliably establishing whether a piece of code is trustworthy.

The prevalent method to tackle this problem consists of testing and code review to check the security of software; however, just like testing for functionality, this can only show the presence of errors but never prove their absence. Moreover, new code is written at a pace that makes any manual verification process hard to implement in the present and impossible to scale in the long run.

In response, a more rigorous approach to software security is needed; ideally, one would ensure that software is secure by construction, with mathematical guarantees establishing that a program leaks no information, where these guarantees are checked mechanically.

However, even though rigorous approaches to software security exist in the academic community, they have yet to see widespread, practical use. This thesis explores approaches to make mathematically rigorous techniques to preventing information leaks more practical.

Section 0.1 provides a cursory look at the topic of language-based security, an approach to software security that makes use of ideas from the programming language community to enforce security properties. Section 0.2 gives a quick overview of the area of information-flow control, which is concerned with formally expressing what it means for programs to be secure, finding techniques that guarantee the security of a program, and proving that they guarantee the desired notion of security. Section 0.3 gives a brief summary of taint tracking and contrasts the approach with information-flow control. Section 0.4 outlines the challenges addressed in this thesis. Section 0.5 summarizes the contributions of the papers comprising this thesis.

0.1 Language-Based Security

A common approach to security in practice involves testing and code review to find security flaws; however, like testing for functionality, this approach cannot prove the absence of security flaws. Language-based security on the other hand is based on providing provable security guarantees by using the semantics of the underlying language: Instead of trying to find security problems in programs, techniques in this area focus on analyzing programs in order to prove that a program is secure, or on automatically preventing insecurities at runtime. Since untrusted programs are granted access to sensitive data, information-flow policies are fine-grained, making language-based techniques an excellent fit for information-flow control.

Broadly speaking, the approaches in language-based security fall into two categories: static techniques analyze a program before it is run and aim to establish whether or not the program is secure for all inputs, while dynamic approaches prevent information leaks as the program is executing. Additionally,
hybrid mechanisms combine static analysis with dynamic techniques.

An example for static approaches are security type systems \cite{320}. In a normal type system, if a program is well-typed, then the program is proven to be free of certain classes of bugs, such as trying to add integers to strings; this is often summarized as “well-typed programs don’t go wrong” \cite{224}. A security type system on the other hand can be used to ensure that all well-typed programs are secure with respect to a security property. Chapter 3 presents a security type system to prevent illicit information flows in web applications.

Dynamic approaches typically modify the semantics of a program in order to prevent leaks of information. For example, if a program sends secret data to the internet, a simple dynamic information-flow enforcement might then terminate the program before the sensitive information is sent out over the network.

0.2 Information-Flow Control

The key feature that makes software security relevant is the amount and type of private information handled by computers today and the fact that they are connected to the internet. However, many programs legitimately need access to both private information and internet connectivity in order to perform their task. As a result, simply not giving a program access to private data is not a viable solution.

For example, a spreadsheet application may be used to manage one’s finances, but also requires internet access to look up stock prices and other live data. If the program is malicious or buggy, it might instead leak one’s financial situation to an attacker on the internet. Chat applications by their nature have access to the messages the user sends and receives, and need internet access to perform their function. However, if the application is malicious, the messages may be sent to attackers in addition to their intended recipients or the application may not encrypt messages correctly before they are transmitted.

The objective of information-flow control is to prevent information flows from sensitive sources, such as local files, to public sinks, such as servers on the internet. In this scenario we assume that the attacker controls public sources, such as data received from the internet, and can observe outputs to public sinks. Moreover, the attacker is assumed to know the code of the program being run, but cannot observe sensitive sinks, such as writing to local files, and cannot observe sensitive inputs to the program, such as user input or contents of local files.

To reason about security of software in a way that provides confidence about a program’s behavior, however we need to express the absence of unwanted information flows needs formally, in unambiguous mathematical terms. The policy of not allowing information to flow from sensitive sources to public sinks is often formalized as noninterference. In this setting, a program is treated as a mathematical object that maps inputs, some of which may be sensitive, to outputs, some of which may be observable by attackers. A program is then considered secure if and only if for two sets of inputs that only differ on their confidential parts, the program always results in the same outputs to the attacker. Otherwise, an attacker can derive information about sensitive inputs by observing the public output of a program. Figure \[\text{Figure 1}\] illustrates this policy.
A simple example setting to describe this policy is to model a program as a function \( \text{prog} : \mathcal{I}_P \times \mathcal{I}_S \rightarrow \mathcal{O}_P \times \mathcal{O}_H \) mapping a pair of public inputs from the set \( \mathcal{I}_P \) and secret input from the set \( \mathcal{I}_S \) to a pair of public outputs in set \( \mathcal{O}_P \) and secret outputs \( \mathcal{O}_H \). Such a program then satisfies noninterference if for the same public input \( i_P \) and any two secret inputs \( i_S \) and \( i'_S \), we have that \( \pi_1(\text{prog}(i_P, i_S)) = \pi_1(\text{prog}(i_P, i'_S)) \) where \( \pi_1 \) refers to the projection a tuple to its first component.

Figure 1: Noninterference

As most interesting program properties, noninterference is undecidable for non-trivial programs \cite{255}. To make matters worse, noninterference is a property about two runs of a program \cite{222} making verification or enforcement more challenging than that of safety or liveness properties commonly considered when verifying functional correctness of programs. As a result, approaches for enforcing noninterference typically compromise on either soundness, i.e. that insecure programs are classified as insecure, or precision, i.e. that secure programs are classified as secure. To still provide meaningful security guarantees, most approaches prioritize soundness over precision at the expense of practicality, since some secure programs can not be successfully verified.

Moreover, since noninterference requires that private information have no influence over any public outputs \textit{at all}, it is sometimes too restrictive. For example, a program handling user logins will necessarily leak information about the stored password, by behaving differently depending on whether or not the user entered the correct password; i.e. it leaks whether the stored sensitive password is equal to the user’s input.

### 0.3 Taint Tracking

Two important categories of information leaks that enforcement mechanisms need to handle are explicit flows, which leak data by directly outputting secret information to an attacker, and implicit flows, which leak information through the control-flow structure of the program. For example, the program

\[
\text{outputTo}(\text{attacker}, \text{secret})
\]

\textsuperscript{1}This glosses over some details such as how to model non-terminating programs, but illustrates the overall idea.
directly sends the value of the secret input stored in variable `secret` to an attacker through an explicit flow, whereas the program

\[
\text{if } secret = 0 \text{ then outputTo}(\text{attacker}, 1) \text{ else outputTo}(\text{attacker}, 2)
\]

leaks the whether the secret input is zero through an implicit flow relying on how control-flow in the program is structured.

While both types of flows leak information, implicit flows are more challenging to track accurately throughout a program, since this involves comparing what happens in two separate branches of a conditional. To avoid imprecision introduced by tracking implicit flows, some approaches only track whether sensitive data directly enters a public sink (such as the `outputTo` function in the previous example). Such approaches are referred to as `taint tracking`. While they do not enforce noninterference, such approaches are often more feasible to use in practice. However, compared to noninterference-based approaches, techniques based on taint tracking often lack formal underpinnings that allow to reason about the guarantees that they provide.

### 0.4 Challenges

**Flexible Policies** While noninterference provides a solid baseline for reasoning about the security of programs, it can be very restrictive. Many useful programs, such as the login program example in Section 0.2, do not satisfy noninterference. While noninterference policies can be relaxed by adding support for declassifying sensitive data [271], declassification policies may be complicated to reason about in practice. Hence, providing a natural and flexible way to specify security policies remains an open problem.

**Formal Guarantees** Despite renewed interest and recent advances in information-flow research [266], the results are largely unused outside of academia. On the other hand, some approaches such as taint tracking have enjoyed considerable success in practice ranging from bug detection to ensuring confidentiality; moreover the technique has been applied to high-level languages and machine code.

However, the actual security policy enforced by practical approaches often remains unclear; as a consequence, evaluating practical security mechanisms in terms of soundness and precision is not possible in an unambiguous way. Hence, providing formal justification and analysis of practical techniques is an important avenue of research for bridging the gap between academic results and industry practice.

**Information-Flow Control for Complex Language Features** Another stumbling block for information-flow control in practice is the gap between minimal languages typically considered in research papers and programming language features used in practice. Extending information-flow control techniques to accommodate settings such as shared memory concurrency, modern databases, and web application scenarios presents an opportunity to make security verification more feasible in practice.
0.5 Contributions

This thesis consists of eight papers (Chapters 1-8). Five papers have been published in peer-reviewed conferences while the remaining three are currently under submission. This section outlines each paper’s contributions and connects its topic to the challenges outlined in the previous sections. Broadly speaking, the papers fall into three categories:

- Formally analyzing of the security guarantees provided by taint trackers, and building more precise enforcement based on this formal characterization.

- Extending information-flow control to challenging language features and policies encountered in practical scenarios. In particular, we consider shared memory concurrency, declassification policies, code interacting with databases, and web applications.

- Bridging the gap between taint tracking and information-flow by investigating intermediate security properties and what trade-offs they offer for real-world code.

The structure of the research is summarized in Figure 2.

Chapters 1 and 2 aim to make security verification more feasible by exploring a more permissive family of security mechanisms known as taint tracking that allows for more practical enforcement mechanisms while nevertheless capturing some realistic use cases:

- Chapter 1 presents a formal characterization of the security property enforced by taint tracking tools. In order to understand the guarantees and ramifications of the taint tracking approach, we need to be able to characterize the attacks that are prevented, or not, by such techniques. In this chapter we present a general security condition capturing which information flows are tracked, and which are ignored, by taint tracking.
• Chapter 2 leverages the security condition from Chapter 5 to construct a general, and in some cases more precise, technique for taint tracking based on the concept of faceted values.

In Chapters 3 through 6, we explore how to extend information-flow control mechanisms to accommodate challenging language features often found in real applications:

• Chapter 3 tackles the problem of information-flow security for programs with shared-memory concurrency requiring expressive declassification policies. We provide a way to accommodate complex thread interaction and controlled information release by decoupling reasoning about functional correctness from information-flow reasoning.

• Chapter 4 investigates how to track information flows in multi-tiered web applications that combine databases with both client-side and server-side code.

• Chapter 5 extends information-flow control conditions and enforcement mechanisms to applications interacting with databases making use of advanced features such as triggers, as well as dynamic security policies that change over time.

• Chapter 6 explores more alternative policies for specifying under what circumstances information can be released by investigating opacity as another property to specify information-flow policies and relating opacity to noninterference.

Chapters 7 and 8 aim to bridge the gap between noninterference-based approaches to information-flow control and techniques such as taint tracking, that are more easily enforceable:

• Chapter 7 presents a security condition that falls between noninterference and definitions based on taint tracking by extending taint tracking to catch another type of dangerous information flows, called observable implicit flows without incurring the same number of false positives as noninterference-based approaches.

• Chapter 8 studies the impact of different types of information flows in the context of real-world JavaScript code to reason about the relative importance of accurately catching different types of information flows.

The rest of this section lists the abstracts of the individual chapters:

0.5.1 Explicit Secrecy: A Policy for Taint Tracking

Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld

Taint tracking is a popular security mechanism for tracking data-flow dependencies, both in high-level languages and at the machine code level. But despite the many taint trackers in practical use, the question of what, exactly, tainting means—what security policy it embodies—remains largely unexplored.
We propose explicit secrecy, a generic framework capturing the essence of explicit flows, i.e., the data flows tracked by tainting. The framework is semantic, generalizing previous syntactic approaches to formulating soundness criteria of tainting. We demonstrate the usefulness of the framework by instantiating it with both a simple high-level imperative language and an idealized RISC machine. To further understanding of what is achieved by taint tracking tools, both dynamic and static, we obtain soundness results with respect to explicit secrecy for the tainting engine cores of a collection of dynamic and static taint trackers.

Statement of Contribution  This paper was co-authored with Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Daniel contributed to the framework. Daniel is responsible for formalizing and proving the results about the explicit secrecy framework, and for formalizing the enforcement mechanisms as well as the soundness proof of the static and dynamic enforcement mechanisms.

Appeared in: Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P), Saarbrücken, Germany, March 2016

0.5.2 Let’s Face It: Faceted Values for Taint Tracking

Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld

Taint tracking has been successfully deployed in a range of security applications to track data dependencies in hardware and machine-, binary-, and high-level code. Precision of taint tracking is key for its success in practice: being a vulnerability analysis, false positives must be low for the analysis to be practical. This paper presents an approach to taint tracking, which does not involve tracking taints throughout computation. Instead, we include shadow memories in the execution context, so that a single run of a program has the effect of computing on both tainted and untainted data. This mechanism is inspired by the technique of secure multi-execution, while in contrast to the latter it does not require running the entire program multiple times. We present a general framework and establish its soundness with respect to explicit secrecy, a policy for preventing insecure data leaks, and its precision showing that runs of secure programs are never modified. We show that the technique can be used for attack detection with no false positives. To evaluate the mechanism in practice, we implement DroidFace, a source-to-source transform for an intermediate Java-like language and benchmark its precision and performance with respect to representative static and dynamic taint trackers for Android. The results indicate that the performance penalty is tolerable while achieving both soundness and no false positives on the tested benchmarks. The key results of this paper have been formalized in Isabelle/HOL.

Statement of Contribution  This paper was co-authored with Musard Balliu, Frank Piessens, and Andrei Sabelfeld. Daniel is responsible for formalizing and proving soundness and precision of the results, as well as the prototype implementation. All authors contributed equally to the writing.
0.5. Contributions


0.5.3 **VERONICA: Verified Concurrent Information-Flow Security Unleashed**

Daniel Schoepe, Toby Murray, Andrei Sabelfeld

Methods for proving that concurrent software does not leak its secrets has remained an active topic of research for at least the past four decades. Despite an impressive array of work, the present situation remains highly unsatisfactory. With contemporary compositional proof methods one is forced to choose between expressiveness (the ability to reason about a wide variety of security policies), on the one hand, and precision (the ability to reason about complex thread interactions and program behaviours), on the other. Achieving both is essential and, we argue, requires a new style of compositional program logic.

We present the first compositional program logic for proving concurrent programs information flow secure that supports high-precision reasoning about a wide range of security policies and program behaviours (e.g. expressive declassification, value-dependent classification, secret-dependent branching). Just as importantly, our approach embodies a new way for engineering such logics that can be re-used elsewhere, called decoupled functional correctness (DFC). DFC leads to a substantially simpler logic, even while achieving this unprecedented combination of features. We demonstrate the virtues and versatility of our approach by verifying a range of example programs, beyond the reach of prior methods. All developments are formalized in Isabelle/HOL.

Statement of Contribution This paper was co-authored with Toby Murray, and Andrei Sabelfeld. Daniel is responsible for developing the enforcement framework, Isabelle formalization, soundness results, and case studies. All authors contributed equally to the writing and overall development of the approach.

Under submission.

0.5.4 **JSLINQ: Building Secure Applications across Tiers**

Musard Balliu, Benjamin Liebe, Daniel Schoepe, and Andrei Sabelfeld

This paper proposes JSLINQ, a framework for writing web applications with end-to-end security guarantees. Modern web applications consist of several tiers, often including server-side code, a database, and client-side JavaScript code. In order to achieve end-to-end security, correct communication between tiers must be ensured. JSLINQ leverages meta-programming facilities in F# and the WebSharper framework to provide a unified language for securely writing the entire web application. A security type system is then used to guarantee noninterference for well-typed programs.
Aside from formal soundness results, we investigate the practicality of this approach with several case studies, such as location-based services and a Battleship browser game, indicating that JSLINQ can handle practical scenarios; this work is therefore a step toward implementing information-flow control in practice.

**Statement of Contribution**  This paper was co-authored with Musard Balliu, Benjamin Liebe, and Andrei Sabelfeld. Daniel contributed to the type inference engine, the language semantics, type system, and the case studies. All authors contributed equally to the writing.


**0.5.5 Information-Flow Control for Database-Backed Applications**

*Marco Guarnieri, Daniel Schoepe, Musard Balliu, David Basin, and Andrei Sabelfeld*

Securing database-backed applications requires tracking information across the program and the database together, since securing each component in isolation may still result in an overall insecure system. Current research extends language-based techniques with models capturing the database’s behavior. Previous work, however, relies on simplistic database models, which ignore security-relevant features that may leak sensitive information.

We propose a novel security monitor for database-backed applications. Our monitor tracks fine-grained dependencies between variables and database tuples by leveraging database theory concepts like disclosure lattices and query determinacy. It also accounts for a realistic database model that supports security-sensitive constructs like triggers and dynamic policies. The monitor automatically synthesizes program-level code that replicates the behavior of database features like triggers, thereby tracking information flows inside the database. We also introduce symbolic tuples, an efficient approximation of dependency-tracking over disclosure lattices. We implement our monitor for database-backed Scala programs and demonstrate its effectiveness on four case studies.

**Statement of Contribution**  This paper was co-authored with Marco Guarnieri, Musard Balliu, David Basin, and Andrei Sabelfeld. Daniel contributed to the formalization and framework and is responsible for the prototype implementation. All authors contributed equally to writing the paper.

*Under submission.*

**0.5.6 Understanding and Enforcing Opacity**

*Daniel Schoepe and Andrei Sabelfeld*

This paper explores opacity, a policy providing more flexible control over what part of a system needs to be kept confidential. Concretely, instead of protecting pieces of data, opacity expresses that properties about the input need to remain secret; for example, a user may not want to disclose whether he is located
in a sensitive area, but is okay with disclosing parts of his location otherwise. Hence, this work falls under the challenge of flexible policies.

The paper provides a general framework for opacity, parametrized in the power of the attacker, and connects opacity to noninterference. Moreover, we explore two dynamic enforcement techniques and provide a proof-of-concept implementation. All theoretical results are formalized using the Isabelle/HOL proof assistant [244].

Statement of Contribution This paper was co-authored with Andrei Sabelfeld. Daniel is responsible for the proofs of the theoretical results, implementation work, and the Isabelle/HOL formalization. All authors contributed equally to writing the paper.


0.5.7 We Are Family: Relating Information-Flow Trackers

Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld

While information-flow security is a well-established area, there is an unsettling gap between heavyweight information-flow control, with formal guarantees yet limited practical impact, and lightweight tainting techniques, useful for bug finding yet lacking formal assurance. This paper proposes a framework for exploring the middle ground in the range of enforcement from tainting (tracking data flows only) to fully-fledged information-flow control (tracking both data and control flows). We formally illustrate the trade-offs between the soundness and permissiveness that the framework allows to achieve. The framework is deployed in a staged fashion, statically embedding a dynamic monitor, being parametric in security policies, as they do not need to be fixed until the final deployment. This flexibility facilitates a secure app store architecture, where the static stage of verification is performed by the app store and the dynamic stage is deployed on the client. To illustrate the practicality of the framework, we implement our approach for a core of Java and evaluate it on a use case with enforcing privacy policies in the Android setting. We also show how a state-of-the-art dynamic monitor for JavaScript can be easily adapted to implement our approach.

Statement of Contribution This paper was co-authored with Musard Balliu and Andrei Sabelfeld. Daniel contributed to the security and enforcement definitions and is responsible for the prototype implementation. All authors contributed equally to writing the paper.

Appeared in: Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS), Oslo, Norway, 2017
Information flow analysis prevents secret or untrusted data from flowing into public or trusted sinks. Existing mechanisms cover a wide array of options, ranging from lightweight taint analysis to heavyweight information flow control that also considers implicit flows. Dynamic analysis, which is particularly popular for languages such as JavaScript, faces the question whether to invest in analyzing flows caused by not executing a particular branch, so-called hidden implicit flows. This paper addresses the questions how common different kinds of flows are in real-world programs, how important these flows are to enforce security policies, and how costly it is to consider these flows. We address these questions in an empirical study that analyzes 56 real-world JavaScript programs that suffer from various security problems, such as code injection vulnerabilities, denial of service vulnerabilities, memory leaks, and privacy leaks. The study is based on a state-of-the-art dynamic information flow analysis and a formalization of its core. We find that implicit flows are expensive to track in terms of permissiveness, label creep, and runtime overhead. We find a lightweight taint analysis to be sufficient for most of the studied security problems, while for some privacy-related code, observable tracking is sometimes required. In contrast, we do not find any evidence that tracking hidden implicit flows reveals otherwise missed security problems. Our results help security analysts and analysis designers to understand the cost-benefit tradeoffs of information flow analysis and provide empirical evidence that analyzing implicit flows in a cost-effective way is a relevant problem.

**Statement of Contribution** This paper was co-authored with Cristian-Alexandru Staicu, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. Daniel contributed to the formalization and framework and the soundness results. All authors contributed equally to writing the paper.

*Under submission.*


[94] Eugene Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In SCALA@ECOOP, 2013.


[139] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In Internet Measurement Conference (IMC), pages 475–488, 2014.


[175] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren’t the droids you’re looking for: Retrofitting android to protect data from imperious applications. In CCS, 2011.


[283] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In S&P 2010, 2010.


