CHALMERS

On the Foundations of Practical Language-Based Security

MAXIMILIAN ALGEHED

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2021

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

On the Foundations of Practical Language-Based Security

MAXIMILIAN ALGEHED

Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden, 2021

On the Foundations of Practical Language-Based Security

Maximilian Algehed

Copyright © 2021 Maximilian Algehed
All rights reserved.

ISBN 978-91-7905-456-4
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 194D
ISSN 0346-718X
This thesis has been prepared using LaTeX.

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
www.chalmers.se

Printed by Chalmers Reproservice
Göteborg, Sweden, April 2021

Thank You Anna

Abstract

Language-based information flow control (IFC) promises to provide programming languages and tools that make it easy for developers to write secure code. Traditionally, research in this field aims to build a variant on a programming language or system that lets developers write code that gives them strong guarantees beyond the potential memory- and type-safety guarantees of modern languages. However, two developments in the field challenge this paradigm. Firstly, backwards-compatible security enforcement without false alarms promises to retrofit security enforcement on code that was not written with the enforcement mechanism in mind. This has the potential to greatly increase the applicability of IFC enforcement to legacy and mobile code from untrusted sources. Secondly, library-based security, a technique by which IFC researchers provide a software library in an established language whose programming interface gives the same guarantees as a stand-alone IFC tool for developers to use promises to do away with specialized IFC languages. This technique also has the potential to increase the applicability of IFC enforcement as developers no longer need to adopt a whole new language to get security guarantees.

This thesis makes contributions to both these recent developments that come in two parts; the first part concerns enforcing secure information flow without introducing false alarms while the second part concerns the correctness of using libraries instead of fully-fledged IFC programming languages to write secure code.

The first part of the thesis makes the following contributions:

1. It unifies the existing literature, in the form of Secure Multi-Execution and Multiple Facets, on security enforcement without false alarms by introducing Faceted Secure Multi-Execution.

2. It explores the unique optimisation challenges that appear in this setting. Specifically, mixing multi-execution and facets means that unnecessarily large faceted trees give rise to unnecessary executions in multi-execution and vice verse. This thesis proposes optimisation strategies that can overcome this hurdle.

3. It proves an exponential lower bound on black-box false-alarm-free enforcement and new possibility results for false-alarm-free enforcement of a variant of the noninterference security condition known as termination insensitive noninterference.

4. It classifies the special cases of enforcement that is not subject to the aforementioned exponential lower bound. Specifically, this thesis shows how and why the choice of security lattice makes the difference between exponential, polynomial, and constant overheads in multi-execution.

In short, the first part of the thesis unifies the existing literature on false-alarm-free IFC enforcement and presents a number of results on the performance of enforcement mechanisms of this kind.

The second part of the thesis meanwhile makes the following contributions:

1. It reduces the trusted computing base of security libraries by showing how to implement secure effects on top of an already secure core without incurring any new proof obligations.

2. It shows how to simplify DCC, the core language in the literature, without losing expressiveness.

3. It proves that noninterference can be derived in a simple and straightforward way from parametricity for both static and dynamic security libraries. This in turn reduces the conceptual gap between the kind of security libraries that are written today and the proofs one can write to prove that the libraries ensure noninterference.

In short, the second part of the thesis provides a new direction for thinking about the correctness of security libraries by both reducing the amount of trusted code and by introducing improved means of proving that a security library guarantees noninterference.

Keywords: Security, Programming Languages, Secure Multi-Execution, Parametricity.

List of Publications

This thesis is based on the following papers:

[A] T. Schmitz, **M. Algehed**, A. Russo, C. Flanagan, "Faceted Secure Multi Execution". *25th ACM Conference on Computer and Communications Security* (CCS), 2018, Toronto, Canada

[B] **M. Algehed**, A. Russo, C. Flanagan, "Optimising Faceted Secure Multi-Execution". *The 32nd Computer Security Foundations Symposium* (CSF), 2019, Hoboken, New Jersey, USA

[C] **M. Algehed**, C. Flanagan, "Transparent IFC Enforcement: Possibility and (In)Efficiency Results". *The 33rd Computer Security Foundations Symposium* (CSF), 2020, Online, Distinguished Paper Award

[D] **M. Algehed**, C. Flanagan, "Multi-Execution Lattices Fast and Slow". *Unpublished*, 2020

[E] **M. Algehed**, A. Russo, "Encoding DCC in Haskell". *ACM SIGSAC Workshop on Programming Languages and Analysis for Security* (PLAS), 2017, Dallas, Texas, USA

[F] **M. Algehed**, "A Perspective on the Dependency Core Calculus". *ACM SIGSAC Workshop on Programming Languages and Analysis for Security* (PLAS), 2018, Toronto, Canada

[G] **M. Algehed**, J.-P. Bernardy, "Simple Noninterference from Parametricity". *24th ACM SIGPLAN International Conference on Functional Programming* (ICFP), 2019, Berlin, Germany

[H] **M. Algehed**, J.-P. Bernardy, C. Hritcu "Dynamic IFC Theorems for Free!". *To Appear in the 34th Computer Security Foundations Symposium* (CSF), 2021

Statement of Contributions

Faceted Secure Multi Execution Thomas and I shared the majority of the technical work and the writing equally. I was mainly responsible for the implementation and evaluation of the Multef tool and case studies. The initial prototype implementation was done by Alejandro, while subsequent rewrites of the framework and re-structuring to greatly simplify the presentation were performed by me.

Optimising Faceted Secure Multi-Execution Alejandro and I jointly came up with the idea of doing tree-rewriting. The idea of limiting views during computation came from my implementation of a chat server in Multef. The technical development, including proofs, implementation, and type-setting were done by me. The majority of the writing was done by me, Alejandro and Cormac provided useful feedback and input on the entire writing process.

Transparent IFC Enforcement: Possibility and (In)Efficiency Results The original idea, model, and the majority of the proof effort was done by me. Cormac contributed valuable proof insight, questions, and feedback on the presentation of the paper.

Multi-Execution Lattices Fast and Slow The original idea, definitions, and technical development was done by me. Cormac contributed the same kind of valuable proof insight, questions, and feedback on the presentation of the paper as in the previous paper.

Encoding DCC in Haskell It was Alejandro Russo's idea to encode DCC in Haskell. The key idea of using type families was mine. The technical development was performed by me. Both authors contributed to the writing, with Alejandro's primary role being to provide feedback on my writing and structure.

A Perspective on the Dependency Core Calculus This is a single author paper, I did all the work on my own.

Simple Noninterference from Parametricity The idea behind the paper and the technical development were both mine. Jean-Philippe contributed the key insight of using Sigma types and performed the Agda mechanisation of the core parametricity proof. The mechanisation of the translation of DCC into Agda was done by both authors jointly.

Dynamic IFC Theorems for Free! The idea behind the paper and the core technical development were both mine. Catalin and Jean-Philippe both contributed key technical and presentation insight. All three authors contributed to the mechanisation and writing.

Acknowledgments

I am grateful for the love and support I have received from all sides while writing this thesis. Thank you Mary for helping me answer my most important research question: Can I graduate with a PhD? Of course, in order to help me answer this one difficult question you also helped me answer so many other questions about self esteem, confidence, and kindness. If it weren't for you I would not have managed to write this thesis and I would not be the person I am today. You have made me smarter, more confident, and happier than I was when I first came to Chalmers. I hope that soon you will get all the calm and uncluttered days that you deserve.

Many people also deserve love and gratitude for saving me from myself, my loving, caring, and intelligent parents Anders and Jessica, my wonderful grandparents, Marianne, Kenneth, and Monica, and my frighteningly clever brother Albin. I love you all.

Anna-Maria, you have helped me get through so many dark days and so much stress. You are the most fun, intelligent, kind, and dare I say cute person I know and I love you to the moon and back. There are many years to come and I look forward to spending them with you.

Andréas, thank you for all the wonderful days and nights on and off the water and thank you for so many years of friendship, advice, patience, and fun. I hope we will go on many more of the kind of ill-conceived, sometimes dangerous, but always brilliantly executed and fun adventures that have always provided both of us with much needed release.

I also want to thank the many teachers who took an angry young boy under their wing and set him on the path to become the person I am today. It would have been very easy for Tommy, Bosse, Lovisa, Johan, and Johan to write a younger me off as a lost cause. I'm forever grateful to all of you for not doing that.

Another person who deserves a big thank you for being a long running friend and a persistent source of both inspiration and amusement is Paul.

Next I want to thank Koen and Alejandro. Koen, although you tell me you turned me away the first time I came to you and wanted to do research (an incident that I honestly can not remember) you have, since then, always encouraged me and given me sane advice. Thank you Alejandro for taking me under your wing and teaching me how to just get on with it and turning ideas into papers. You have been instrumental to many parts of this thesis.

I also want to thank the many people at Chalmers who have helped me navigate the process of obtaining my degree, including Aarne, Agneta, Nir, and Clara who have all been most helpful and patient.

Next I want to thank my long-standing office mate and friend Sòlrùn for many uplifting conversations and fun adventures. I also want to thank Alexander and Pauline for always having my back and making sure I stay sane and don't get too much important work done in one day. Likewise I want to thank Iulia for forcing me to, on occasion, take

things seriously and for the amusement that has resulted from neither of us doing so. A special thank you also to Sandro whose patience, compassion, and kindness has played a big role in helping me figure out what I want to do with my life. In fact, there are too many people at Chalmers for me to name here who have all been good at keeping me from getting too much work done in a day and who have all contributed to making the department a great place to work, I'm forever grateful.

I would also like to thank the many wonderful and colourful people I met at Cornell. Thank you Andrew for hosting me, making it an interesting stay, and showing me how a great group of people can work together on so many problems at the same time. Thank you Ethan for being the most helpful and welcoming person one could ever wish to have around and for being so much fun. Thank you Matthew for being a great friend and thinker. Thank you Rolph for being both a caring individual and a true pleasure to work with and for letting me in on your many interesting ideas and thoughts. Likewise, thank you Shir, Josh, Haobin, Drew, Andy, Jonathan, Rachit, Molly, and Soumya (in no particular order) for making my stay both friendly and interesting.

A final big thank you should also go to my many co-authors without whom I would have never finished this thesis. Thank you Tommy for being gracious and helpful when I came late to the party. Thank you Cormac for teaching me how to write and always challenging me to think, both about computers and about the world. Thank you Jean-Philippe and Catalin for helping me do such interesting work and having the patience to help me understand even the most simple things.

There are also people outside of academia who deserve much gratitude for our professional and personal relationships that have taught me to deal with many of the stranger things in the world of work. Niklas, Jenny, and Lukas deserve particular thanks for many days of hard, but amusing, work on a project that I think we all feel great pride in. Likewise, I want to thank Marcus for the work and fun we've shared.

Finally, I want to acknowledge that there are so many more people who deserve my gratitude than I have the space to name here. If you are reading this, chances are that you have played some important role in making me the person I am today and making this thesis happen. Thank you.

This thesis was completed with support from the Wallenberg AI, Autonomous Systems and Software Program (WASP) and the Barbro Osher Pro Suecia Foundation.

Contents

Abstract i

List of Papers iii

Statement of Contributions v

Acknowledgements vii

1 Introduction 3
- 1.1 Backwards Compatible Security 4
- 1.2 Scalable, Provable, and Principled Security by Construction. 6

2 Beyond the State of the Art of Backwards-Compatibility 9

3 Security by Embedding Tools 13

References 17

I Transparent IFC Enforcement 25

A Faceted Secure Multi-Execution A1
- 1 Introduction A3
- 2 Background A5
- 3 A Unifying Multi Execution Framework A7
  - 3.1 Functional core A7
  - 3.2 Faceted values A8

3.3 FIO computations A8
3.4 Building side-effectful computations based on faceted values A9
3.5 Supported multi-executions approaches A11
3.6 Formal semantics A12
4 Termination Insensitive Security Guarantees A16
5 Fair Scheduling A18
6 Termination Sensitive Security Guarantees A19
7 Decentralized Labels A20
7.1 Disjunction Category Labels A21
8 Implementation A22
8.1 Basic structures A22
8.2 Executor commonalities A23
8.3 MF executor A24
8.4 Continuations and SME A25
8.5 FSME executor A26
9 Evaluation A26
10 ProtectedBox A29
10.1 Labeling policy A29
10.2 Performance A30
11 Related work A31
12 Conclusions A32
Appendix A - Semantics and Proof Sketches A33
Appendix B - Implementation A34
Appendix C - FSME (switching) executor A37
References A39

B Optimising Faceted Secure Multi-Execution B1
1 Introduction B3
2 Background B6
2.1 Security Lattices B6
2.2 Faceted Values B7
2.3 Residuated Lattices and Galois Connections B9
3 Data-Oriented Optimisations B9
3.1 Constructing Residuated Lattices B13
3.2 Residuation of DC-labels B14
3.3 Context-aware optimisations B15
4 Computation-Oriented Optimisation B18
4.1 Core Calculus B18
4.2 Removing unnecessary views B21
5 Case Studies B23
5.1 Data-Oriented Optimisation B23

5.2 Computation-Oriented Optimisation B25
6 Related Work B28
7 Conclusions B29
Appendix A - Syntax and Semantics B29
References B34

C Transparent IFC Enforcement: Possibility and (In)Efficiency Results C1
1 Introduction C3
2 An Extensional Framework for Secure Information Flow C5
2.1 Programs with Labeled Inputs and Outputs C5
2.2 Secure Programs and Termination Criteria C7
2.3 Enforcement Mechanisms C10
2.4 Multi-Execution C11
2.5 The State of Transparent Enforcement C13
3 Multi-Execution for Decentralised Lattices C14
4 Termination Insensitive Noninterference is Transparently Enforceable . . C17
5 Transparent Enforcement is Multi-Execution C19
6 Efficient Black-Box Enforcement is Impossible for Decentralised Lattices C23
7 Future Work C25
8 Related Work C26
9 Conclusion C27
Appendix A - Omitted Proofs C28
A.1 - Proofs showing that $C(S)$ partitions the lattice C28
A.2 - Properties of MEF C29
A.3 - Level Assignments C30
A.4 - Lemmas for Theorems 6 and 7 . . . . . C30
References C31

D Multi-Execution Lattices Fast and Slow D1
1 Introduction D3
2 Review of the Multi-Execution Framework D5
3 Great and Small D11
3.1 Product Lattices D15
3.2 Sum Lattices D16
3.3 Exponential Lattices D17
3.4 k-Truncated Powersets D18
4 Fast and Slow D19
5 Through the Looking Glass D20
5.1 Galois Connections D20
5.2 Execution Time of $\mathrm{MEF}^{F \dashv G}$ D25
5.3 Finding Galois Connections D26
6 Empirical Results D28

7 Related Work . . . D30
8 Conclusions . . . D31
Appendix A - Great and Small . . . D31
Appendix B - Fast and Slow . . . D36
Appendix C - Through the Looking Glass . . . D36
References . . . D39

II Soundness of Security Libraries 45

E Encoding DCC in Haskell E1
1 Introduction . . . E3
2 Background . . . E4
3 Embedding DCC in Haskell . . . E8
3.1 Revisited example . . . E10
4 An alternative formulation . . . E12
5 Side-Effects . . . E15
5.1 Outputs . . . E16
5.2 Label creep . . . E19
6 Combining Effects . . . E22
6.1 Error handling . . . E22
6.2 State . . . E23
6.3 I/O . . . E24
7 Related work . . . E25
8 Final remarks . . . E26
Appendix A - Automatic relabeling . . . E27
Appendix B - Structure for outputs . . . E27
Appendix C - Label creep . . . E28
Appendix D - State . . . E28
References . . . E29

F A Perspective on the Dependency Core Calculus F1
1 Introduction . . . F3
2 The Dependency Core Calculus . . . F3
3 SDCC, A Simpler Core . . . F6
4 An Equally Expressive Calculus . . . F7
5 A Simple Haskell Implementation . . . F10
6 Related and Future Work . . . F12
7 Conclusions . . . F12
References . . . F13

G Simple Noninterference from Parametricity **G1**
1 Introduction . . . G3
2 Parametricity . . . G5
3 Intuition Behind the Proof of Noninterference from Parametricity . . . G10
4 Noninterference from Parametricity . . . G11
5 Shallow Embedding of Dependency Core Calculus . . . G15
6 Deep Embedding of Dependency Core Calculus . . . G21
7 Implementation in Haskell . . . G24
8 Related Work . . . G26
9 Conclusion and Future Work . . . G27
References . . . G28

H Dynamic IFC Theorems for Free! **H1**
1 Introduction . . . H3
2 Dynamic IFC as a Library . . . H5
3 Parametricity and Data Abstraction . . . H10
4 Two Proofs of Noninterference . . . H15
4.1 Noninterference for Faceted Values . . . H15
4.2 Noninterference for Core LIO . . . H18
5 Transparency . . . H25
6 Related and Future Work . . . H28
7 Conclusion . . . H29
References . . . H30

Overview

CHAPTER 1

Introduction

The state of the art in computer security is problematic. Computer systems that handle sensitive data are vulnerable to being attacked by anyone from amateurs and skilled hobbyists[1] to organized groups like nation states and crime syndicates.

The kind of tools we use to build and maintain computer systems are inadequate to guarantee that the systems are secure. For example, due to the Heartbleed bug [1], HTTPS servers using the popular OpenSSL [2] library, that implements encrypted communication, were vulnerable to attackers reading data from users' encrypted connections. The root cause of Heartbleed was a simple memory vulnerability; the attackers could craft a special message that caused OpenSSL to read data beyond the bounds of an array and send it to the attacker. In effect, however, this vulnerability was caused by the fact that, for whatever historical and pragmatic reason, the programmer who wrote OpenSSL used a memory-unsafe programming language, C, that provided them with little to no help in finding and eliminating an out-of-bounds read of sensitive data.

This thesis is about the theoretical foundations of a new group of technologies based on programming languages and formal methods that try to address problems like this in computer security. My goal is to combine theory and practice in an attempt to both address *urgent problems* with solutions that are *backwards compatible* and create *principled tools* that make it possible for future systems to be built in a way that guarantees *security by construction*.

With the goal of addressing the general problem that our tools are inadequate to ensure the confidentiality of data and the integrity of systems, the research in this thesis fits into a broader push to use formal methods and programming languages technology to

[1] Yes, some people do bad things for fun

bring the toolkit available to engineers up to date with the challenges we are facing. Specifically I consider three main challenges that need to be addressed.

- Firstly, we need tools to formally express – in languages readable by both computers and humans – the *security policy* of a system: what the system is and is not intended to do with and to data.
- Secondly, we need tools to *enforce* such security policies on systems. These tools should make the system adhere to the policy and will be needed both during design and development of a system and during the system's operation.
- Finally, we need to address both of the challenges above for both *existing* as well as *future* systems.

One approach to solving problems of this kind is *language-based security* [3]. In this field, we provide programming languages, abstractions, type systems, and runtime systems that give concrete security guarantees to the programmer. To help address these challenges, this thesis presents work in two main areas, backwards compatible- and secure-by-construction language-based security.

1.1 Backwards Compatible Security

The first major security challenge to address is securing the systems that are already running today. Historically, when a new security vulnerability is found in a system, one of two things happens.

On the one hand, if the issue affects a single piece of software or system, a patch is typically issued that fixes the problem and users are instructed to upgrade to a newer version. Anyone in possession of a modern smartphone will be familiar with the never-ending stream of notifications saying that the latest version of the operating system and "critical security updates" is available for download.

On the other hand, if a whole class of security issues is found, we typically see solutions that target the "root cause" of the vulnerability. For example, attack techniques like stack-smashing [4] and Return-Oriented Programming (ROP) [5], whereby the attacker overwrites a return address on the execution stack to get control over execution, have been addressed by tools like StackGuard [4] and Address Space Layout Randomization (ASLR).

While security patches are a tedious necessity, these latter solutions are great. They address the techniques used to exploit memory vulnerabilities and make them more difficult to use. Furthermore, enabling ASLR will most likely not cause any existing, secure, software to change its behaviour. In other words, technologies like ASLR are backwards compatible.

However, the problem is that most such solutions, like ASLR, are ad-hoc. Instead of ensuring that the target program is actually secure, ASLR only guarantees that the

target program can not be attacked using a specific type of attack. Consequently, there is still room for new attacks that work around ad-hoc mechanisms by exploiting other weaknesses of the system [6]. This allows the cycle to continue with new ad-hoc defensive mitigations that plug these new holes [7].

Unfortunately, this is not an isolated issue in how memory vulnerabilities are addressed. Rather, it happens across the board. For example, the prevalence of Cross Site Scripting (XSS) attacks [8]–[10] on the web, whereby an attacker executes code in the user's browser without their consent, have prompted a number of more-or-less ad-hoc counter measures aimed at, among other things, making XSS more difficult [11]. Unfortunately, these ad-hoc measures can be circumvented by new attacks like DOM-based XSS [9], [10] and so the problem of XSS persists.

In short, the state of the art for backwards-compatible security is unsustainable. New attacks will continue to surface so long as our defences are designed to guarantee that a particular attack does not work, rather than defending against all attacks at once.

That said, the state of the art *does* make it more difficult for attackers to exploit programs in a real, quantifiable, way [6], [7], [12] without breaking programs that are "well-behaved". This outlines the shape of a more general, and hopefully more (though not entirely) permanent, solution to the problem. We want mechanisms that address security issues by guaranteeing some concrete, extensional (meaning caring only about the *what* of a program, not the *how*), security policy of the program, rather than only ruling out specific attacks, without breaking already secure programs.

The point about extensionality is particularly important to this goal. Ad-hoc mechanisms can give formal guarantees about the internal behaviour of programs; ASLR makes it provably difficult for the attacker to exploit certain ROP-like vulnerabilities [13]. However, ASLR does not guarantee that the program does not leak secrets. For example, ASLR will not stop you from running a shell script that sends your private SSH key to the secret police by email.

In this sense, ASLR gives you only *intensional* guarantees about *how* your program computes, not *what* effects it has on the outside world. This is typical for the ad-hoc defensive measures; when you only try to stop a specific attack you only end up caring about *how*. *Extensional* guarantees meanwhile are about the opposite. They care about what your program does to the world around it, not how it does it.

The contributions in Part I of this thesis are aimed at finding the theoretical opportunities and limits of one particular approach, known as multi-execution [14], [15], to providing backwards-compatible, extensional, security guarantees. We will return to this topic in Chapter 2.

1.2 Scalable, Provable, and Principled Security by Construction.

Looking forward to building future secure systems we see that modern programming languages and tools are ill-suited for specifying and *ensuring* security properties in programs. The most advanced tools and systems that are *used in practice* are aimed at ensuring simple properties like crash-freedom [16]. In the best case, programmers use a memory-safe language that defends against the kind of errors in memory management that lead to some security vulnerabilities [17]–[19]. However, while memory-safety ensures basic integrity properties that make life harder for attackers, it falls short of ruling out insecure behaviour altogether.

We need programming languages that provide programmers with abstractions and tools that guarantee absence from attacks. Programmers should be able to give and enforce security policies that specify how information in their program can be used, and what data may influence what control flow and side-effects of their code.

The Information Flow Control (IFC) literature promises to do precisely this [3]. In this line of work, researchers produce programming languages that allow the programmer to specify an information flow policy for their program that the language then guarantees, up-to some notion of what constitutes "information flow".

For example, the simple policies are of the form

> Secret information does not influence the result computed by a deterministic programs.

which codifies the assumption that information only flows through the actual values a program outputs whereas the policy

> Secret information does not influence the execution time of programs.

codifies a stronger notion that information can flow through timing. Variants on this latter policy are often used in contexts like cryptography [20], where timing-based attacks can pose a serious threat to both confidentiality and integrity [21].

Being able to guarantee properties of this kind and being able to precisely specify what does, and does not, constitute "information flow" makes for a promising research field. However, the typical solutions in this literature fall short of being generally applicable for one main reason: Every combination of a notion of information flow and enforcement style is implemented as its own stand-alone tool or language.

For example, we have specialised type systems in the form of stand-alone tools like Jif [22] for Java and FlowCaml [23] for Caml. We also have specialised runtime environments for dynamic enforcement, like JSFlow [24] for JavaScript.

It is entirely understandable that the field has developed in this direction. Designing one's own language or variant of a language with a stand-alone tool offers enough freedom for the researcher to focus on controlling information flow, soundness conditions,

and what makes for useful yet secure abstractions. The programming languages literature contains more examples of subfields that have developed this way before eventually settling on a small number of core languages that become more popular [25].

One concern is that this approach is unsuited for practical applications as it forces programmers to adopt a new language for every part of their system that needs a different kind of IFC approach. Furthermore, programmers also need to trade-off their IFC-related concerns with other concerns such as their own competence with that language or the tooling ecosystem surrounding their language of choice.

There are two possible solutions to this problem. The first option is to create a once-and-for-all language for writing secure code for a given domain [22], [26]. The second option is to come up with some "language-internal" method, like libraries or sandboxing of untrusted code that can give the necessary guarantees without forcing the programmer to switch programming language [27], [28].

With the diversity of modern programming languages it seems difficult to realise the first solution, which is why this thesis focuses on the second. That is not to say that there are no insights to be gained from pursuing work in the first direction, it allows the researcher more freedom to design the programming language the way they want. Finding library-based or language-internal solutions to information flow problems on the other hand constrains the researcher to work within the confines of the language. This naturally provides challenges and opportunities both in practice and in theory.

One should also note that there is an interplay between these two approaches. As we will see in Chapter 3, the library-based approaches can not be applied to an arbitrary programming language, the language of choice needs to have a number of key features. This means that work on library-based security also needs to feed into programming language design in general to be widely applicable. In other words, there is no panacea for writing secure code and we need to further develop our programming languages to help programmers regardless of what approach one takes.

CHAPTER 2

Beyond the State of the Art of Backwards-Compatibility

To address security issues in a principled and backwards compatible manner, the first part of this thesis presents work on enforcing policies without introducing false alarms [14], [15], [29]–[46]. In theory, some *information flow security* policies, like *noninterference*: a program's secret outputs can not depend on its public inputs, can be enforced without introducing false alarms, so-called *transparent* enforcement, using a mechanism called *Secure Multi-Execution* (SME) [14].

SME, depicted in Figure 2.1, works by running the program p once for every security level in the input. In Figure 2.1 we have two security levels public, or L, and secret, or H. The execution of p associated with H is given both the secret and the public input and is responsible for producing only the secret output. The execution of p associated with L meanwhile is given only the public input and either no secret input or a default secret input, depending on the precise setting one is interested in, and is responsible for producing only the public output.

Intuitively speaking, the result of executing p under SME is secure; by construction none of the secret information in the input can influence the public information in the output. This soundness condition is known as *noninterference*; a program is noninterfering if given two inputs that differ only in their secret parts, the public outputs remain the same.

The relationship between p and SME go two ways. SME guarantees noninterference, and if p is noninterfering, then SME applied to p is functionally equivalent to p; they produce the same output on the same inputs, provided that both runs of p terminate.[1]

[1] Cognoscenti will note that there is no end to the caveats surrounding SME and termination, and the interested reader will find that we discuss this at some length in the papers.

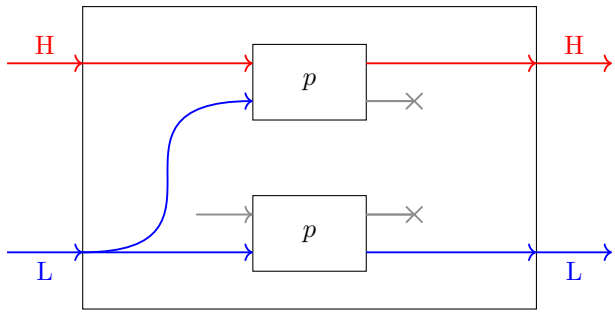

Figure 2.1: Secure Multi-Execution of the program p for two security levels.

The secret outputs are the same in both runs, and it doesn't matter to the public output of p whether or not it gets the *real* secret input or the default (or non-) input in the public run from SME. This completeness-like property is known as *Transparency*.

Transparency makes SME and related approaches like Multiple Facets (MF) [15], which we will see later, promising. While plenty of other mechanisms exist that enforce noninterference-like properties [3], they suffer from an unmanageable number of false alarms [47], [48] or undecidability [49], [50]. This stops them from being applied to existing systems in a backwards compatible manner. Transparent enforcement approaches, meanwhile, have the advantage that, in theory, they don't suffer from this issue at all.

However, my collaborators and I have shown that multi-execution will introduce unmanageable, exponential, overheads in runtime. This happens both in theory [32] and practice [31]. Furthermore, our theoretical results apply not just to SME, but to *any* such transparent enforcement mechanism that is black-box, i.e. does not work by reading the source-code of the program under enforcement. Finally, I have proposed mechanisms for getting around this overhead [31] in proof of concept implementations and shown that, in theory, they have efficacy for future practical case studies.

Contributions and Future Work

Concretely, the high-level contributions of each of the papers in this part of the thesis are the following:

Faceted Secure Multi Execution In this paper we show that different *intensional* approaches to multi-execution, a "fine-grained" SME approach similar to that of Devriese and Piessens [14] and the MF approach of Austin and Flanagan [15], can be unified into something called Faceted Secure Multi-Execution (FSME). The main advantage of this is that it allows us to study trade offs between the time and memory overhead of each of these approaches, and tune the overhead by having "more or less" of one or the other technique for a given program. The

main drawback of this paper is that because we are interested in the *theory* of how SME and MF can be unified, the system we propose to study it does not work for "any piece of code", but rather for code written using our framework. However, this is an irrelevant restriction from the point of view of the lessons learned in the paper. The point of the paper is not to actually secure third-party code (which would be difficult if the code had to be written in our language), but rather to show the correspondence between SME and MF. In other words, this paper provides a *semantic domain* for multi-execution rather than a tool for securing code. The most important consequence of this work is to show that not only can SME and MF be unified, a previously unresolved piece of folklore, but doing so gives rise to opportunities for new trade-offs to be made in future designs of multi-execution enforcement mechanisms. For example, we show that there is a performance trade-off between time- and memory-overheads and that by carefully mixing SME and MF one can get a "best of both worlds" mix of low overheads in both time and memory, for some programs.

Optimising Faceted Secure Multi-Execution In this paper we show how to optimise the implementation and application of FSME. There are two lines along which we optimise, *data-oriented* and *computation-oriented*. In the former, we re-write the representations of data that looks different to different security levels (so-called faceted values) using a series of rewrite rules that we conceive of and prove sound. In the latter, we use the insight, that will re-appear later in the thesis, that not all computations produce output at all security levels to introduce an optimisation to FSME computations by simply not executing some views of the program under enforcement. The data-oriented optimisations reduce overhead in a black-box manner, the computation-oriented ones do not and require some analysis or insight from a programmer to be applied.

Transparent IFC Enforcement: Possibility and (In)Efficiency Results In this paper we show a number of things about transparent IFC enforcement in general. This includes upper and lower bounds on execution time overhead for black-box enforcement, possibility and impossibility results, and a polynomial-time reduction from any enforcement mechanism to a mechanism based on multi-execution.

Multi-Execution Lattices Great and Small In this paper we build on the work in the previous paper and show that while the worst-case execution time overhead for multi-execution is indeed exponential, this overhead is determined by the number of security levels the program considers, and how they can be combined.

The first part of this thesis presents the following contributions made by myself in collaboration with my many co-authors:

- We unify Multiple Facets and Secure Multi-Execution under a scheme we call Faceted Secure Multi-Execution (FSME) implemented as the Multef framework.
- We implement ProtectedBox, a prototype case study in using FSME as a tool for building systems where third-party plug-ins can be integrated cleanly without having to worry that the third-party code is secure.
- We show how to optimise FSME by rewriting faceted trees and limiting the number of executions necessary for multi-execution to work transparently and securely.
- We present a new model for transparent information flow control in which we show that:
  1. Some previously unaccounted for versions of noninterference can be transparently enforced.
  2. There is no black-box efficient transparent enforcement mechanism for any variant of noninterference.
- We show how the shape of the security lattice that is used by multi-execution matters to the worst-case runtime of transparent enforcement.

Put differently, this part of my thesis is a deep-dive into the theoretical limits on backwards-compatible, transparent, enforcement mechanisms for noninterference-like security properties. The papers in this part of the thesis focus primarily on the efficiency of transparent IFC enforcement, but there are also a few novel possibility results. What this thesis does not present on the other hand is the engineering work necessary to solve the problem of transparent enforcement for practical domains.

There are a number of avenues for future work following this part of the thesis.

- Implement a general-purpose mechanism for transparent enforcement of JavaScript code in web browsers that integrates the work on trade-offs in this thesis with the seminal FlowFox SME-browser [51].
- Work out the bounds on overhead in time, for white-box enforcement mechanisms, and space, for both white- and black-box enforcement mechanisms.
- Systematically investigate how mechanisms for transparent IFC compose, and to what extent the pre-condition that “the target program is noninterfering” can be loosened to show that multi-execution preserves “partially correct slices” of the target program.
- Combine multi-execution with other mechanisms for transparent enforcement of security properties to get a more flexible, efficient, framework for transparent enforcement.

CHAPTER 3

Security by Embedding Tools

The part of language-based security that enables the secure-by-construction approach that I consider in this thesis is about providing security guarantees beyond notions like memory- and type-safety. Specifically, I'm interested in *embedding* both *static analyses* and *semantics* that provide information flow guarantees as *libraries* or *frameworks* in existing languages. The approach of embedding what is traditionally a separate type system or semantics for a language as a library, rather than a stand-alone tool, provides the opportunity to make these technologies viable for practical use without having to implement a new language or force programmers to use a specialized compiler.

Put more concretely, there are two main benefits of this approach over building stand-alone languages that provide the kind of guarantees that are necessary to build secure software, flexibility and culture.

Firstly, programmers need flexibility in what kind of security guarantees their programming language provides [52]. The range of possible languages and guarantees is relatively large, from guarantees of simple end-to-end noninterference [3] to guarantees about side-channels based on schedulers [53], probabilistic security guarantees [54], and any number of different notions of what makes for a principled way to make secrets public and otherwise downgrade sensitive information [55]–[57]. This means that programmers need to be able to pick and choose their guarantee to fit the threats their code is going to encounter. Security libraries have the potential to make this picking and choosing less problematic as the same programming language can play host to multiple different libraries [27], [28]. If, as an added bonus, the same language can host libraries that provide different guarantees then programmers have a wider range of choices than if a given language could provide only one kind of security guarantee.

Secondly, as I'm sure readers of this thesis are aware, programming languages come with culture and legacy. Every developed programming language used for real-world purposes today has an ecosystem of libraries, tools, compilers, editors, versioning systems, and knowledgable programmers. Rather than throwing this culture away, or rebuilding it for a new programming language that provides "just the right security guarantees", a more fruitful way forward is to piggy-back on this culture by enriching it with security libraries that help programmers get those same guarantees without having to abandon their language [28], provided that the language can play host to such a library.

One of the main benefits of these libraries is that they allow us to take the insights from the language-based security literature [58] and put them to work to build systems with relatively little effort. For example, popular libraries and frameworks for the strongly typed Haskell programming language [27], [28], [59] have been used in practise to implement practical systems securely, like the "Build It Break It Fix It" security contest [59], [60] and the Hails web framework [61].

Security libraries work because they use three key language-features of their host language: abstract types, strong typing, and control over side-effects. Abstract types mean that secrets, or computations that compute secrets, are forced to adhere to the interface of the library. This adherence is in turn enough to keep secrets from leaking using a strong type system that controls side-effects. The latter, control over side-effects, enforces that the programmer uses the library or framework's controlled interface to the operating system to communicate with the outside world and avoid information leaks. In other words, these libraries rely on sophisticated properties and systems in their host language.

My work on these security libraries focuses on techniques for proving that the libraries are sound [62], [63]. In this setting, soundness means that the library guarantees some security policy, often but not exclusively in the form of *noninterference*, as discussed above, for any client code that uses the library.

I have developed new techniques that narrow the gap between existing soundness proofs, that tend to be based on a number of simplifying assumptions, and the actual implementation of a library. Existing proofs work by assuming that the library and the host programming language taken together can be equivalently considered a new programming language that has all the features, and guarantees, that the library provides out-of-the-box. In contrast, my proofs work by re-using the meta theory of the host programming language to reason about the implementation of the library.

This shift of perspective is important for two reasons. Firstly, it allows us to make fewer assumptions about the way that the implementation of a library interacts with the host programming language. The technique does away with the *assumption* that the library uses encapsulation techniques like type abstraction or private fields correctly and turns it into a *proof obligation*.

Secondly, it allows us to re-use large parts of the established meta theory for the host programming language to reason about the implementation of the library. This is

important, as it allows our proofs to more easily scale to larger libraries. For example, in a recent paper my collaborators and I shortened the state-of-the-art proof for the popular LIO security library [28], which forms the foundation of many more applied libraries, by an *order of magnitude* by using our technique while also *covering more of the library* than the original proof [63].

However, due to the techniques we are using in these proofs, parametricity for pure type systems [64], we are limited in our ability to reason about fully-fledged languages and libraries. Put simply, this is because the necessary theory that we rely on has not been extended to cover the kind of languages that practical IFC libraries and frameworks, like LWeb [59] and Hails [61], are implemented in. That said, this theory is being actively developed [65], [66].

Contributions and Future Work

Concretely, the high-level contributions of each of the papers in this part of the thesis are the following:

Encoding DCC in Haskell In this paper we show how to encode the Dependency Core Calculus of Abadi et al. [67] in Haskell. This exercise alone is not particularly surprising, but we then show how to implement a number of secure effects on top of this DCC "implementation" using monad transformers. Importantly, the security of the underlying DCC implementation means that these transformer-based effects are secure *by construction*. While we do not prove that in this paper, the last two papers in this thesis make this point precise and establish it more rigorously.

A Perspective on the Dependency Core Calculus In this paper I show how to simplify DCC to get rid of one of the particularly complicated primitives in the language by building an equally expressive but simpler language I call SDCC and showing that DCC and SDCC are equally expressive.

Simple Noninterference from Parametricity In this paper we develop the foundations for reasoning faithfully about library-based information flow control. We use the parametricity meta theory for pure type systems of Bernardy et al. [68] to prove noninterference for an embedding of DCC in the Calculus of Constructions. The embedding is morally equivalent to the embedding we use above and the focus on the paper is primarily on specifically proving noninterference without appealing to a model of the embedding, as has been done in previous work that embeds IFC as libraries [27], [28].

Dynamic IFC Theorems for Free! In this paper we extend the methods developed in "Simple Noninterference from Parametricity" to reason about libraries for *dynamic* information flow control. These proofs are somewhat more involved than the proofs in the previous paper, but the main strategy is the same. The

important result in this paper is that not only does parametricity let us reason *faithfully* about how an IFC system is embedded as a library, it also lets us reason efficiently. The proofs in this paper are an order of magnitude shorter than proofs in previous work while making fewer assumptions and omissions.

In this part of the thesis, I make the following contributions:

- We show how to embed the Dependency Core Calculus in Haskell using the security-as-a-library approach.
- I show that the Dependency Core Calculus can be simplified without modifying its expressiveness.
- We show how to use parametricity to prove noninterference for a DCC-like library.
- We show how noninterference for the library gives rise to noninterference for DCC.
- We show how to prove noninterference using parametricity for two dynamic IFC libraries.

The main contribution of this part of the thesis to the field of language-based security is that my co-authors and I establish the parametricity proof technique as a viable and scalable option for proving soundness of security libraries. My hope is that in the future this will allow us to scale up security-as-a-library to be a viable option for large-scale systems *without compromising on soundness proofs*.

There are a number of high-level avenues for future work for anyone interested in building on the work in this part of the thesis:

- Continue scaling up the proof methods to work with automated and semi-automated theorem provers that, unlike the current Agda mechanisation, would allow our proofs to scale to libraries of thousands of lines rather than a few hundred.
- Scale the security-as-a-library approach and parametricity proof methods to languages that are used in practice.
- Find new domains for security-as-a-library and find ways to extend the proofs to work in these domains.
- Weaken the three requirements on the host language of the security library either by finding new mechanisms by which to provide security libraries (for example via some form of sandboxing) or by finding lightweight ways to add the necessary capabilities to more languages.

References

[1] *CVE-2014-0160.* Available from MITRE, CVE-ID CVE-2014-0160.

[2] OpenSSL Software Foundation, *OpenSSL*, `https://www.openssl.org/`, Accessed 2020-11-12.

[3] A. Sabelfeld and A. C. Myers, "Language-based information-flow security", *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[4] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting systems from stack smashing attacks with StackGuard", in *Linux Expo*, 1999.

[5] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications", *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012.

[6] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR", in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–13.

[7] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses", in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 249–266.

[8] K. et al., *Cross site scripting (XSS)*, `https://owasp.org/www-community/attacks/xss/`, Accessed 2020-11-11.

[9] A. Klein, "Dom based cross site scripting or XSS of the third kind", *Web Application Security Consortium, Articles*, vol. 4, pp. 365–372, 2005.

[10] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild.", 2019.

[11] *Content security policy reference*, `https://content-security-policy.com/`, Accessed 2020-11-12.

[12] V. Ganesh, S. Banescu, and M. Ochoa, "Short paper: The meaning of attack-resistant systems", in *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, 2015, pp. 49–55.

[13] ——, "The meaning of attack-resistant systems", *arXiv preprint arXiv:1502.04023*, 2015.

[14] D. Devriese and F. Piessens, "Noninterference through secure multi-execution", in *Security and Privacy (SP), 2010 IEEE Symposium on*, IEEE, 2010, pp. 109–124.

[15] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow", in *ACM Sigplan Notices*, ACM, vol. 47, 2012, pp. 165–178.

[16] R. Milner, "A theory of type polymorphism in programming", *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[17] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018, ISBN: 1593278284.

[18] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson, "Report on the programming language haskell: A non-strict, purely functional language version 1.2", *SIGPLAN Not.*, vol. 27, no. 5, pp. 1–164, May 1992, ISSN: 0362-1340.

[19] K. Arnold, J. Gosling, D. Holmes, and D. Holmes, *The Java Programming Language*. Addison-wesley Reading, 2000, vol. 2.

[20] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography", in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1267–1279.

[21] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems", in *Annual International Cryptology Conference*, Springer, 1996, pp. 104–113.

[22] A. Banerjee and D. A. Naumann, "Secure information flow and pointer confinement in a java-like language.", in *CSFW*, vol. 2, 2002, p. 253.

[23] V. Simonet and I. Rocquencourt, "Flow caml in a nutshell", in *Proceedings of the first APPSEM-II workshop*, 2003, pp. 152–165.

[24] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in javascript and its APIs", in *Proc. of the ACM Symposium on Applied Computing (SAC '14)*, ACM, Mar. 2014.

[25] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: Being lazy with class", in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007, pp. 12–1.

[26] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, “Jsflow: Tracking information flow in javascript and its apis”, in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ACM, 2014, pp. 1663–1671.

[27] M. Vassena, A. Russo, P. Buiras, and L. Waye, “Mac a verified static information-flow control library”, *Journal of Logical and Algebraic Methods in Programming*, vol. 95, pp. 148–180, 2018, ISSN: 2352-2208.

[28] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, “Flexible dynamic information flow control in haskell”, in *ACM Sigplan Notices*, ACM, vol. 46, 2011, pp. 95–106.

[29] M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz, “A better facet of dynamic information flow control”, in *WWW’18 Companion: The 2018 Web Conference Companion*, 2018, pp. 1–9.

[30] T. Schmitz, M. Algehed, C. Flanagan, and A. Russo, “Faceted secure multi execution”, in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 1617–1634.

[31] M. Algehed, A. Russo, and C. Flanagan, “Optimising faceted secure multi-execution”, in *Proceedings of the 32nd IEEE Computer Security Foundations Symposium (CSF)*, IEEE, 2019, pp. 1–115.

[32] M. Algehed and C. Flanagan, “Transparent IFC enforcement: Possibility and (in)efficiency results”, in *Proceedings of the 33rd IEEE Computer Security Foundations Symposium (CSF)*, IEEE, 2020, pp. 65–78.

[33] M. Jaskelioff and A. Russo, “Secure multi-execution in haskell”, in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, Springer, 2011, pp. 170–178.

[34] T. Schmitz, M. Algehed, C. Flanagan, and A. Russo, “Faceted secure multi execution”, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 1617–1634.

[35] M. Algehed, A. Russo, and C. Flanagan, “Optimising Faceted Secure Multi-Execution”, in *Proc. of the 2019 32nd IEEE Computer Security Foundations Symp.*, ser. CSF ’19, IEEE Computer Society, 2019.

[36] M. Algehed and C. Flanagan, “Transparent ifc enforcement: Possibility and (in)efficiency results”, in *2020 IEEE Symposium on Computer Security Foundations*, IEEE, 2020.

[37] D. Zanarini, M. Jaskelioff, and A. Russo, “Precise enforcement of confidentiality for reactive systems”, in *2013 IEEE 26th Computer Security Foundations Symposium*, IEEE, 2013, pp. 18–32.

[38] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, “Flowfox: A web browser with flexible and precise information flow control”, in *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, 2012, pp. 748–759.

[39] ——, "Secure multi-execution of web scripts: Theory and practice", *Journal of Computer Security*, vol. 22, no. 4, pp. 469–509, 2014.

[40] M. Ngo, F. Piessens, and T. Rezk, "Impossibility of precise and sound termination-sensitive security enforcements", in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 496–513.

[41] M. Ngo, F. Massacci, D. Milushev, and F. Piessens, "Runtime enforcement of security policies on black box reactive programs", in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 43–54.

[42] T. Pfeffer, T. Göthel, and S. Glesner, "Efficient and precise information flow control for machine code through demand-driven secure multi-execution", in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, ACM, 2019, pp. 197–208.

[43] K. Micinski, D. Darais, and T. Gilray, "Abstracting faceted execution", in *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, IEEE, 2020, pp. 184–198.

[44] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: Fine-grained, declassification-aware, and transparent", *Journal of Computer Security*, vol. 24, no. 1, pp. 39–90, 2016.

[45] I. Boloşteanu and D. Garg, "Asymmetric secure multi-execution with declassification", in *International Conference on Principles of Security and Trust*, Springer, 2016, pp. 24–45.

[46] N. Bielova and T. Rezk, "Spot the difference: Secure multi-execution and multiple facets", in *European Symposium on Research in Computer Security*, Springer, 2016, pp. 501–519.

[47] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em", in *International Conference on Information Systems Security*, Springer, 2008, pp. 56–70.

[48] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript", *arXiv preprint arXiv:1906.11507*, 2019.

[49] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs", in *International Symposium on Formal Methods*, Springer, 2011, pp. 200–214.

[50] G. Barthe, P. R. D'argenio, and T. Rezk, "Secure information flow by self-composition", *Mathematical Structures in Computer Science*, vol. 21, no. 6, p. 1207, 2011.

[51] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, “Secure multi-execution of web scripts: Theory and practice”, *Journal of Computer Security*, vol. 22, no. 4, pp. 469–509, 2014.

[52] I. Bastys, F. Piessens, and A. Sabelfeld, “Prudent design principles for information flow control”, in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, 2018, pp. 17–23.

[53] S. Zdancewic and A. C. Myers, “Observational determinism for concurrent program security”, in *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.*, IEEE, 2003, pp. 29–43.

[54] D. Volpano and G. Smith, “Probabilistic Noninterference in a Concurrent Language”, *J. Computer Security*, vol. 7, no. 2–3, Nov. 1999.

[55] O. Arden, J. Liu, and A. C. Myers, “Flow-limited authorization”, in *2015 IEEE 28th Computer Security Foundations Symposium (CSF)*, Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2015, pp. 569–583.

[56] E. Cecchetti, A. C. Myers, and O. Arden, “Nonmalleable information flow control”, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 1875–1891.

[57] S. Zdancewic and A. C. Myers, “Robust declassification.”, in *csfw*, Citeseer, vol. 1, 2001, pp. 15–23.

[58] A. Sabelfeld and A. C. Myers, “Language-based information-flow security”, *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[59] J. Parker, N. Vazou, and M. Hicks, “Lweb: Information flow security for multi-tier web applications”, *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 75, 2019.

[60] A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, “Build it, break it, fix it: Contesting secure development”, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 690–703.

[61] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, “Hails: Protecting data privacy in untrusted web applications.”, in *OSDI*, 2012, pp. 47–60.

[62] M. Algehed and J.-P. Bernardy, “Simple noninterference from parametricity”, *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–22, 2019.

[63] M. Algehed, J.-P. Bernardy, and C. Hritcu, “Dynamic IFC theorems for free!”, in *To Appear in the Proceedings of the 34th IEEE Computer Security Foundations Symposium (CSF)*, 2021.

[64] J.-P. Bernardy, P. Jannson, and R. Paterson, “Proofs for free - parametricity for dependent types”, *Journal of Functional Programming*, vol. 22, no. 2, pp. 107–152, 2012.

[65] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, “Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning”, *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 637–650, 2015.

[66] S. O. Gregersen, J. Bay, A. Timany, and L. Birkedal, “Mechanized logical relations for termination-insensitive noninterference”, 2020.

[67] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, “A core calculus of dependency”, in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1999, pp. 147–160.

[68] J.-p. Bernardy, P. Jansson, and R. Paterson, “Proofs for free: Parametricity for dependent types”, *Journal of Functional Programming*, vol. 22, no. 2, pp. 107–152, 2012.