Flexible and Practical
Information-Flow Control

Daniel Schoepe

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

Göteborg, Sweden 2016
Flexible and Practical Information-Flow Control

DANIEL SCHOEPE

© 2016 Daniel Schoepe

Technical Report 149L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Language-based Security

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY and GÖTEBORG UNIVERSITY
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2016
Abstract

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 that is processed by computing systems is not disclosed to third parties without the user’s permission; i.e. to prevent unwanted flows of information. Since most approaches to information-flow control have not seen widespread use in practice, this work explores flexible policies and enforcement techniques to guarantee that information is not leaked by a program. The thesis consists of three parts:

The first chapter explores opacity, a security policy for protecting sensitive system properties, motivated by location privacy and privacy-preserving aggregation scenarios. We present a general, parametric framework for opacity and relate it to noninterference. Moreover, we present two approaches to enforcement: a dynamic monitor making use of SMT solving, and a blackbox sampling-based approach based on the random testing tool QuickCheck.

The second chapter discusses taint tracking, a popular security mechanism for tracking data-flow dependencies, which is widely used for both high-level languages and machine code. However, 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. We illustrate our approach by instantiating explicit secrecy to both, a high-level imperative language and machine code. Additionally, we prove soundness with respect to explicit secrecy for the cores of dynamic and static taint trackers.

Lastly, we present JSLINQ, a framework providing end-to-end information-flow control for multi-tiered web applications; i.e. web applications consisting of a database, server-side code, and client-side JavaScript code. To prevent information flows at component boundaries, we leverage homogeneous meta-programming features in F# to provide a unified language for programming all components. We present a security type system for a core of F# and prove that all well-typed programs are noninterfering. We evaluate our approach using various case studies indicating that JSLINQ is suitable for implementing practical web applications.
ACKNOWLEDGMENTS

This thesis comprises my work of the last two years, and, like any such endeavor, could not have been accomplished without the help and support of others.

Firstly, I want to thank my supervisor Andrei without whose excellent and helpful support, none of this work would have been possible. Besides the fruitful collaboration and supervision, Andrei always went out of his way to help me in my career and development; and by that I do not just mean telling me about great restaurants and coffee places in Gothenburg. I also want to thank my co-supervisor Musard for the great collaboration and useful input. Dave and Wolfgang also deserve gratitude for their help over the last two years.

I am also very grateful for the wonderful working environment and colleagues at the department. It is very rewarding to have colleagues who are not just people to work with, but also friends. I would like to thank Pablo, Alexander, Per, Irene, Víctor, Musard, Inari, Herbert, Simon, Raúl, Evgeny, Daniel, Hamid, Mauricio, John, Luciano, Bart, Marco, Aljoscha, Bassel, and everyone I forgot to mention here for the countless after-works, discussions, barbecues, parties, fikas, and for making Chalmers such an enjoyable environment.

While working in research is a very rewarding experience, it is important not to forget that there is also a life outside of academia. I owe a debt of gratitude to Maria, Eike, Jin, Елена, Fabian, Brian, Marie, Fotini, Charlotte, and many others for all the fun and support over the years and for putting up with me for this long. Ytterligare tack till Maria för att öva svenska med mig, även om det ibland måste vara lite smärtsamt att höra sitt språk manglas.

Lastly, I also want to thank my family for their support and encouragement over the course of my life. I could not have gotten to where am I today without their help.

Given how rewarding my first two years of working at Chalmers have been so far, I look forward to another three enjoyable years.
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. We use computers and other devices to talk to friends, keep track of our calendars, take pictures, watch movies, read books and newspapers, and to handle countless other tasks.

As a result, a significant portion of our day-to-day lives and of our personal information is controlled by software; billions of lines of code that need to function correctly for us to go about our day. In the process, we make a significant amount of sensitive data available to the software, ranging from our location over search queries and private messages to pictures and videos. 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 sure that an application does not send sensitive data to a third party? A malicious application could leak confidential files on the hard drive, passwords, or even credit card numbers entered in a web application.

Disclosing private data handled by applications can have a severe impact on a person’s life. If a credit card number is sent to a scammer, the victim will lose money. A stolen social security number can give rise to identity theft. Companies may lose profits if internal data is leaked, as in the case of the Sony hack in 2015 [6]. Disclosing a person’s political views, private conversations, religious beliefs, or sexual orientation might cause them to lose their job or
impact their social life. In extreme cases, such as dissidents living under an oppressive regime, disclosing private data may even lead to a loss of human lives.

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 underlie our society. Ever since Edward Snowden’s revelations \cite{3}, we know that even Western governments steal their citizen’s information on a large scale \cite{7}, ostensibly in an effort to combat terrorism, although evidence that mass surveillance is in any way helpful in that regard has yet to surface \cite{4, 2}. 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. Since one of the methods used for stealing data is to spread malicious or backdoored software to the users’ devices \cite{5, 1}, this illustrates the need for a way of reliably establishing whether a piece of code is trustworthy.

Moreover, the numerous reports of successful attacks that permeate the media show that current industry practices are not sufficient to create trustworthy software. The prevalent method is to use 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. In response, a more rigorous approach to software security is needed; ideally, one would ensure that software is secure \textit{by construction}, with mathematical guarantees establishing that a program leaks no information.

However, even though powerful attackers abound, efforts to improve the security and trustworthiness of software from the academic community have yet to see widespread use. This thesis aims to make a step toward more practical approaches for arriving at software that is provably secure.

Section \ref{0.1} 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 and with finding techniques that guarantee the security of a program. Section \ref{0.2} 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 \[0.3\] outlines the challenges addressed in the work of the thesis. Section \[0.4\] summarizes the contributions of the papers comprising this thesis.

0.1 Information-Flow Control

The key feature that makes software security relevant is the amount of private information handled by computers today. Such information can be damaging to individuals if disclosed; in the case of dissidents in autocratic countries, malicious software can even endanger human lives.

However, many programs legitimately need access to private information in order to perform their task. For example, a word processor may be used to write sensitive documents, but at the same time requires internet access to check for security updates; if the program is malicious, it can send out the sensitive documents to a third party on the internet. 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 sensitive sources.

However, this policy needs to be expressed formally, in unambiguous mathematical terms, in order to reason in a reliable way about the information-flow security of programs. 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 inputs that only differ on their sensitive parts, the program always produces the same outputs. Otherwise, an attacker can derive information about sensitive inputs by observing the public output of a program. Figure \[1\] illustrates this policy.

As most interesting program properties, noninterference is undecidable for non-trivial programs. To make matters worse noninterference is a property about two runs of a program \[8\]. This
makes verification or enforcement of it more challenging than that of safety or liveness properties commonly considered in the area of program verification. 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. Due to the resulting false positives, techniques for information-flow have not been widely deployed in practice.

Moreover, since noninterference requires that private information have no influence over any public outputs at all, it is sometimes too restrictive. For example, a program handling user logins will necessarily leak information about sensitive data, namely 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.

Chapters 1 and 2 explore more flexible policies that allow for certain, controlled leaks of information and are, in some scenarios, amenable to more practical enforcement techniques.

Chapter 3 presents an enforcement technique for noninterference that makes it more practical to write real-world web applications while maintaining information-flow guarantees.

0.2 Language-Based Security

The common approach in practice to security involves testing and code review to find security flaws; however, like testing for func-
0.3. 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 0.1 do not satisfy noninterference. While noninterference policies can be relaxed by adding support for declassifying sensitive data [13], 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 [12], 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 in Practice** Aside from the challenges outline above, there sometimes exists a mismatch between the type of programs considered in the literature, often short snippets in a minimal core of programming language, and software in the real world, often complex systems written in different languages and distributed across different machines. Applying advances in information-flow research to real-world scenarios presents another interesting challenge.

### 0.4 Contributions

This thesis consists of three papers published in peer-reviewed conferences. This section outlines each paper’s contributions and connect the topic to the challenges outlined in the previous sections.
0.4. Contributions

0.4.1 Understanding and Enforcing Opacity

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 [10].

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. Both authors contributed equally to writing the paper.


0.4.2 Explicit Secrecy: A Policy for Taint Tracking

In this paper, we investigate taint tracking, a popular mechanism for tracking direct data flows through a problem; while this approach is widely used in practice, it falls short of enforcing noninterference, as it ignores leaks resulting from the program’s control flow; in fact, what formal policy is enforced by taint tracking remains largely unexplored. We propose explicit secrecy, a general framework for expressing security with respect to explicit flows, i.e. flows that leak data directly as opposed to leaking data through the control-flow behavior of the program.

We instantiate the framework for a high-level imperative language and RISC-style machine code. Moreover, we model the cores of dynamic and static taint trackers and prove it sound for explicit secrecy.

This work addresses both the need for flexible policies as well as investigating formal guarantees of practical mechanisms.
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 prove of the static enforcement.

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

0.4.3 JSLINQ: Building Secure Applications across Tiers

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 theoretical results and the case studies. All authors contributed equally to the writing.

Bibliography


CHAPTER ONE

UNDERSTANDING AND ENFORCING OPACITY

Daniel Schoepe and Andrei Sabelfeld

Abstract. This paper puts a spotlight on the specification and enforcement of opacity, a security policy for protecting sensitive properties of system behavior. We illustrate the fine granularity of the opacity policy by location privacy and privacy-preserving aggregation scenarios. We present a framework for opacity and explore its key differences and formal connections with such well-known information-flow models as noninterference, knowledge-based security, and declassification. Our results are machine-checked and parameterized in the observational power of the attacker, including progress-insensitive, progress-sensitive, and timing-sensitive attackers. We present two approaches to enforcing opacity: a whitebox monitor and a blackbox sampling-based enforcement. We report on experiments with prototypes that utilize state-of-the-art Satisfiability Modulo Theories (SMT) solvers and the random testing tool QuickCheck to establish opacity for the location and aggregation-based scenarios.
1.1 Introduction

This paper puts a spotlight on the specification and enforcement of opacity [11, 46, 39, 14], a security policy for protecting sensitive properties of system behavior. Intuitively, a predicate on system behaviors is opaque if for any behavior that satisfies the predicate, there is another behavior that is indistinguishable by the attacker but where the predicate no longer holds.

Scenarios for opacity Opacity is a natural policy in many scenarios. As of 2011, Flickr’s geoprivacy settings [25] include geofences for geotagging policies in sensitive areas, e.g., as depicted in Figure 1.1, no one can see the location of the photos that a user tags in a geofenced area (e.g., the user’s home). Scenarios like Flickr’s aim at improving privacy of location-based services (LBS), an important area with much recent attention and progress [42, 62]. We elaborate on variations of the scenario where an LBS is required to protect users’ location, but only when they are within sensitive areas. Note that there are many scenarios beyond location privacy, e.g., regulating how the result of a health test can be released depending on its outcome, in a similar fashion as location release that depends on the location.

Figure 1.1: Flickr’s geofences

Suppose $hX$ and $hY$ store the user’s potentially sensitive (or high) Cartesian location coordinates. We assume the coordinates are static, as tracking users over time would require additional protection. Say, a clinic, representing a sensitive location, occupies a rectangular area with the corners $(200, 50)$ and $(400, 150)$. Consider the scenario of an LBS that outputs on a public (or low) channel, with the goal of privacy-preserving output.
In Program 1, when the user is inside the clinic, the program reports a default location (100, 200) and otherwise the user’s real location. Can the common security definitions directly support the scenario? Noninterference [30] and its knowledge-based analogues [36, 22] allow no flow from high to low whatsoever and thus wrongfully reject the intuitively secure program. Declassification [58] and knowledge-based release [4] models for intentional information release also have difficulties with the scenario. Fundamentally, declassification is often about what can be released [19, 41, 54, 55, 28, 43], while opacity is, conversely, about what must be kept secret. Declassification policies can be categorized along the dimensions of what information is released, when and where in the system the release takes place, and by whom the release is controlled [58]. When focusing on protecting secret inputs, closest to opacity are the what policies since they are concerned with separating secrets from non-secrets in the provided input. These partial release [19, 41, 54, 55, 28, 43] policies specify what is released by splitting the domain of secrets into subdomains and only protecting secret variation within the subdomains. When it comes to specifying partial release, the subdomains can be expressed as kernels for escape hatch expressions [55], but since in our scenarios the released information depends on the location, escape hatch expressions would need to be nearly as complex as the program itself! We will come back to the relation to declassification in Section 1.8.

On the other hand, opacity is a natural policy for this scenario. The property to protect is whether the user is inside the clinic. If the user is indeed there, the program will return the default location (100, 200). Rightfully, there is another run of the program that originates outside the clinic (in (100, 200)) and produces the same observation as the original run.
Although the scope of this paper limits opacity to protecting information about input environments, opacity in general allows arbitrary behavioral properties of a system to be protected, and not only the values of secret inputs.

Program 2 is similar to Program 1 except a (pseudo)random outside location is returned whenever the user is inside the clinic. It makes it more difficult for the attacker to learn anything about the user’s location. When observing the output, the attacker is unable to deduce whether it is the user’s real location outside the clinic or a randomly generated location while the user is actually inside. As we elaborate later, this program satisfies symmetric opacity in the sense that both the sensitive predicate (whether the user is inside the clinic) and its negation are protected.

Program 3 directly models Flickr’s geoprivacy policy. The user’s location is simply suppressed if within the sensitive area and output normally otherwise. When observing the output, the attacker
learns that the user is outside the sensitive area, which is safe. As elaborated later, this scenario connects to progress-insensitive security [2, 3, 10].

```c
/* Program 4 - Statistics aggregation */
int[10] hHasDisease; /* Declare hHasDisease as array */
i := 0;
result := 0;
while (i < 10) {
    result := result + ((hHasDisease[i] > 0) ? 1 : 0);
i := i + 1;
}
out L result;
```

In a different scenario, Program 4 iterates over an array `hHasDisease` (whose size is public) checking for patients diagnosed with a disease and aggregate the total of positive cases. The sensitive predicate to protect is whether a given patient is diagnosed positively or negatively. Common security definitions require a special treatment for such corner cases as when there is only one patient and when the count of the positively diagnosed patients is zero or the same as the total count. On the other hand, opacity covers corner cases by design because it directly focuses on the key property (predicate) to be hidden. In similar vein, opacity is a good fit for the electronic voting scenario, allowing to reveal the total count of votes for a given candidate without revealing the individual votes of any given voter.

**Research questions** With the above building the intuition for opacity in a variety of scenarios, the paper seeks to answer fundamental questions on understanding and enforcing opacity. Given the rich literature on security definitions and specifications [44, 26, 52, 56], the question is whether such common concepts as noninterference [30], knowledge-based security [36, 22, 4], and declassification [58] relate to opacity. If so, what is the exact relation? How does the relation depend on the power of the attacker?

An important unexplored problem is how to assure opacity. This leads to several questions. How can opacity be enforced? Is whitebox or blackbox enforcement more appropriate or perhaps both can be useful? If so, how well can such techniques help with the outlined scenarios?

**Contributions** To answer the questions above, we present a
framework for opacity, spell out the differences, and formalize and machine-check the relation to noninterference and knowledge-based security in a fashion parametric in the attacker power, including progress-insensitive, progress-sensitive, and timing-sensitive attackers (Section 1.2). We instantiate the formal connection results to batch-job (Section 1.3) and interactive (Section 1.4) settings. We relate opacity to the what dimension of declassification, as captured by the model of delimited release [55] (Section 1.5). We present two approaches to enforcement, a whitebox dynamic monitor and a blackbox sampling-based mechanism, both established sound (Section 1.6). We build prototypes that utilize state-of-the-art Satisfiability Modulo Theories (SMT) solvers Z3 [20] and CVC4 [7] as well as the random testing tool QuickCheck [16] and report on our experiments for the scenarios of location privacy and statistics aggregation (Section 1.7). Finally, the paper offers a discussion of related work (Section 1.8) and concluding remarks (Section 1.9).

1.2 Framework

We characterize the connection between opacity and common security definitions in a general, language-independent way, to be instantiated to a batch-job setting in Section 1.3 and an I/O setting in Section 1.4. Our results are parameterized in the attacker’s power. Further, we relate opacity to knowledge-based security.

Assume $E$ to be the set of environments. Denote the set of possible results of a program by $R$. A configuration $\langle c, e \rangle$ is a pair consisting of a command $c \in C$ and an environment $e \in E$. Assume that there is an evaluation relation $\langle c, e \rangle \Downarrow r$ relating a configuration $\langle c, e \rangle$ with a result $r \in R$.

Note that this allows for nondeterministic semantics. However, we assume that the secrets that should be kept confidential are about initial environments, not about nondeterministic components in outputs. Focusing on the protection of initial environments enables us to formally connect opacity with the common security definitions. As foreshadowed earlier, hiding general system properties is one of the advantages of opacity, opening promising avenues for future work, such as reasoning about program obfuscation.
1.2.1 Opacity

Drawing on the original work on opacity [11, 46, 39, 14], we present a general framework for opacity in a semantics-based setting. For a predicate $\varphi$ to be opaque for a command $c$, any environment satisfying $\varphi$ must correspond to another, observably equivalent, environment that does not satisfy $\varphi$ while producing observably equivalent outputs. Intuitively this states that an attacker can never be certain that $\varphi$ holds given public input and output of a program.

The notion is parametric in equivalence $\overset{i}{\sim}$ on environments and relation $\overset{o}{\sim}$ on results. The intuition is that $\overset{i}{\sim}$ expresses what information the attacker can observe about environments while $\overset{o}{\sim}$ captures what parts of the result are visible to the attacker.

**Definition 1** ($\text{Op}(c, \overset{i}{\sim}, \overset{o}{\sim})$). $\varphi$ is opaque for command $c$, equivalence relation $\overset{i}{\sim}$ and relation $\overset{o}{\sim}$ (written $\varphi \in \text{Op}(c, \overset{i}{\sim}, \overset{o}{\sim})$) if and only if whenever $\langle c, e_1 \rangle \downarrow r_1$ and $e_1 \in \varphi$, then there exist $e_2, r_2$ such that $\langle c, e_2 \rangle \downarrow r_2 \land e_1 \overset{i}{\sim} e_2 \land r_1 \overset{o}{\sim} r_2 \land e_2 \not\in \varphi$.

This definition allows for the attacker to learn that $\varphi$ does not hold, based on observing public behavior of a program. In some scenarios, it should also be kept secret that the predicate does not hold, i.e. an attacker should neither be able to infer that $\varphi$ is not satisfied.

Note that the implication trivially holds if $e_1 \not\in \varphi$, e.g the empty set is trivially opaque for any program $c$ and any relations $\overset{i}{\sim}$ and $\overset{o}{\sim}$.

Recall the scenario of keeping secret whether the user is located in a clinic. Program 1 resolves this by outputting the same outside coordinate when the user is actually inside. Consider an initial memory $m_1$ where $m_1(hX) = 5$ and $m_1(hY) = 5$. In this case, the attacker observes the output $(5, 5)$. Based on the program the attacker can then infer that the user must be actually located at position $(5, 5)$: if the user had been inside the sensitive area, the observable output would have been $(100, 200)$. The following definition captures scenarios where leakage of this form is not acceptable:

**Definition 2** ($\text{SOp}(c, \overset{i}{\sim}, \overset{o}{\sim})$). $\varphi$ is symmetrically opaque for command $c$, equivalence relation $\overset{i}{\sim}$ and relation $\overset{o}{\sim}$ (written $\varphi \in \text{SOp}(c, \overset{i}{\sim}, \overset{o}{\sim})$) if and only if whenever $\langle c, e_1 \rangle \downarrow r_1$ and $e_1 \in \varphi$, then there exist $e_2, r_2$ such that $\langle c, e_2 \rangle \downarrow r_2 \land e_1 \overset{i}{\sim} e_2 \land r_1 \overset{o}{\sim} r_2 \land e_2 \not\in \varphi$.
mand c and relation $\sim$ (denoted by $\varphi \in SOp(c, i\sim, o\sim)$) if and only if $\varphi \in Op(c, i\sim, o\sim)$ and $\overline{\varphi} \in Op(c, i\sim, o\sim)$, where $\overline{\varphi}$ denotes the complement of $\varphi$.

As demonstrated above, this condition is violated by Program 1. To achieve symmetric opacity, one solution is to output random coordinates outside of the sensitive area as in Program 2, discussed in detail in Section 1.7.

In order to reason about opacity of a predicate for a single run of a program, we also consider a single-run version of opacity:

**Definition 3.** A predicate $\varphi \subseteq \mathbb{M}$ is opaque for command $c$, environment $e_1$, and result $r_1$ (written $\varphi \in Op(c, i\sim, o\sim, e_1, r_1)$) if and only if

$$e_1 \in \varphi \land \langle c, e_1 \rangle \downarrow r_1 \Rightarrow \exists e_2, r_2. \langle c, e_2 \rangle \downarrow r_2 \land e_2 \sim e_2 \land r_1 \sim r_2 \land e_2 \not\in \varphi$$

This corresponds to the intuition of opacity: When an attacker sees one particular run, he cannot infer whether a predicate holds, since there is another run not satisfying the predicate producing the same observations. Moreover, the following lemma substantiates this intuition:

**Lemma 1.** $\varphi \in Op(c, i\sim, o\sim)$ if and only if $\forall e_1, r_1. \varphi \in Op(c, i\sim, o\sim, e_1, r_1)$.

All proofs have been formalized in Isabelle/HOL and can be found online\(^1\). Pen-and-paper proofs can be found in Appendix 1.A.3.

Additionally, the following properties connect opacity to various logical operations:

**Lemma 2.** If $\varphi \in Op(c, i\sim, o\sim)$ or $\psi \in Op(c, i\sim, o\sim)$, then $\varphi \cap \psi \in Op(c, i\sim, o\sim)$.

**Lemma 3.** If $\varphi \cup \psi \in Op(c, i\sim, o\sim)$, then $\varphi \in Op(c, i\sim, o\sim)$ and $\psi \in Op(c, i\sim, o\sim)$.

\(^1\)http://www.cse.chalmers.se/~schoepe/opacity
The reverse implications do not hold in general: Consider, \( \psi = \overline{\varphi} \). Since \( \varphi \cup \varphi \) cannot be opaque, the converse of Lemma 3 does not hold if \( \varphi \) and \( \overline{\varphi} \) are opaque. Similarly, a counter-example for the converse of Lemma 2 can be constructed by choosing \( \varphi \) and \( \psi \) such that \( \varphi \cap \psi = \emptyset \) with \( \varphi \) or \( \psi \) opaque.

Moreover, Lemma 3 can be used to extend enforcement mechanisms to several predicates at once, albeit at the loss of precision.

1.2.2 Noninterference

We now present a definition of noninterference [30], also parametric in equivalence \( \sim \) on environments and relation \( \sim \) on results.

**Definition 4** \((NI(\sim, \sim))\). \( c \) is noninterferent for equivalence \( \sim \) and relation \( \sim \) (written \( c \in NI(\sim, \sim) \)) if and only if whenever \( \langle c, e_1 \rangle \Downarrow r_1 \) and \( e_1 \sim e_2 \) then there exists a result \( r_2 \) such that \( \langle c, e_2 \rangle \Downarrow r_2 \) and \( r_1 \sim r_2 \).

This definition of noninterference is sufficiently general to capture different flavors of noninterference commonly considered in literature. By varying the relation \( \sim \) we can obtain termination-insensitive [63] and termination-sensitive [64] notions of noninterference in for batch-job programs (Section 1.3) as well as progress-insensitive [2, 3, 10], progress-sensitive [3], and timing-sensitive [56] noninterference for I/O programs (Section 1.4).

By varying \( \sim \), we can express both baseline noninterference and delimited release [55], a notion of declassification (Section 1.5). The results for declassification apply to both batch-job programs and I/O programs.

Recall that noninterference prohibits the example programs (Program 1, Program 2, Program 3, and Program 4), although they are intuitively secure.

1.2.3 Main Lemma

We connect noninterference and opacity by identifying a set of predicates that need to be opaque for a program to satisfy noninterference.

Intuitively, noninterference holds when no secret is leaked through the attacker’s observations of a program run. Conversely, opacity
for one predicate allows leaks of any kind except leaking whether the predicate in question is satisfied. This leads to the following connection: A program satisfies noninterference if and only if it is opaque for “almost all” predicates.

We need to restrict the set of predicates under consideration since a predicate referring only to the actual observations, instead of secrets, can never be opaque: any execution resulting in the same observations will result in either both or none of the runs satisfying the predicate.

As an example, consider the predicate on memories \( \varphi_l = \{ m \mid m(l) = 1 \} \). If a memory \( m_1 \) satisfies \( \varphi_l \), then \( m_1(l) = 1 \). Hence any other memory \( m_2 \) for which \( m_1 \sim m_2 \) holds (i.e. is indistinguishable to the attacker from \( m_1 \)), will also satisfy \( m_2(l) = 1 \) and therefore \( m_2 \in \varphi \). Hence this predicate can never be opaque, not even for a program producing no attacker-visible results (e.g. a program consisting solely of the command \texttt{skip}). Predicates of this form must be ruled out to obtain a connection between noninterference and opacity.

We can express this by requiring that for every class of equivalent environments, the predicate must be violated for at least some memory in that class. Otherwise it cannot possibly state information about secrets, since, within such an equivalence class, all the secrets can vary arbitrarily.

\textbf{Definition 5 (ker(\( \sim \)))}. The kernel of an equivalence relation \( \sim \) is the set of equivalence classes of environments with respect to the relation: \( \ker(\sim) = E / \sim \). The equivalence class of an environment \( e \) is denoted by \( [e]_\sim \).

We say a predicate is informative if it refers to secrets; formally, a predicate \( \varphi \) is informative wrt. some equivalence class for observations on environments \( E \) if and only if \( \varphi \subseteq E \).

We can now provide a precise connection between noninterference and opacity, building on the intuition sketched above.

\textbf{Lemma 4}. For every equivalence \( \sim \) and any reflexive relation \( \simeq \), the following holds:

\[ c \in NI(\sim, \simeq) \iff \forall E \in \ker(\sim). \forall \varphi \subseteq E. \varphi \in Op(c, \sim, \simeq) \]
Intuitively, each predicate required to be opaque in the lemma can be seen as one possible leak of information via the public behavior of the program. If all such predicates are opaque, no such leak is present in the program and hence the program is noninterferent. To illustrate the lemma further, consider the program

\[
\text{if } (h > 0) \{ \text{out L 1} \} \text{ else } \{ \text{out L 2} \},
\]

leaking one bit of information about a secret variable \( h \) using an if-statement.

This program is clearly not noninterferent, since the output produced depends on whether \( h > 0 \) holds. Moreover, the predicate \( \{ m | m(h) > 0 \} \) is not opaque: For an initial memory \( m_1 \) where \( m_1(h) > 0 \), there is no corresponding memory \( m_2 \notin \varphi \) producing the trace \( (1) \).

Moreover, the following lemma establishes that opacity for a smaller set already implies opacity for the set of predicates in Lemma 4:

**Lemma 5.** \( \forall E \in \ker(\sim). \forall e \in E.(E \setminus \{e\}) \in Op(c, \sim, \varnothing) \) if and only if \( \forall E \in \ker(\sim). \forall \varphi \subset E. \varphi \in Op(c, \sim, \varnothing) \).

Intuitively, for every informative predicate \( \varphi \subset E \) for equivalence class \( E \), there must exist some environment \( e \) that does not satisfy \( \varphi \). If the set \( E \setminus \{e\} \) is then opaque, this must yield an equivalent execution starting in environment \( e \), since it is the only environment in this equivalence class not satisfying \( E \setminus \{e\} \).

### 1.2.4 Knowledge-based Characterization

To give further insight into the connection between noninterference and opacity, we also relate opacity and knowledge-based formulations of the attacker’s uncertainty [36, 22, 4].

The basic intuition of the knowledge-based approach is to characterize what the attacker can infer about initial memories in terms of the sets of memories that are possible after seeing a certain result (and the initial values of low variables). We define the set of possible initial memories given some environment \( e_0 \) as well as the set of possible initial memories after observing some result.

**Definition 6.** We define the set of knowledge given an initial environment \( e_0 \) and an equivalence \( \sim \) on environments as \( k(\sim, e_0) = \{e | e \sim e_0\} \).
The knowledge set after observing a run of \( \langle c, e_0 \rangle \) producing result \( r_0 \) is defined by 
\[ k(c, i\sim, \sim, e_0, r_0) = \{ e \mid \exists r. e \sim e_0 \land \langle c, e \rangle \Downarrow r \land r \sim r_0 \}. \]

In terms of knowledge, opacity then holds if there is always a possible initial memory not satisfying \( \varphi \).

**Lemma 6.** \( \varphi \in \text{Op}(c, i\sim, \sim) \) iff \( \forall m_0. \forall r_0 \in R(c, e_0). k(c, i\sim, \sim, e_0, r_0) \cap \varphi \neq \emptyset. \)

where \( R(c, e_0) = \{ r_0 \mid \langle c, e_0 \rangle \Downarrow r_0 \} \).

Intuitively this states that for a predicate to be opaque, the attacker can gain more knowledge as long as he cannot rule out that the predicate is still not satisfied.

### 1.3 Batch-job Programs

To demonstrate the applicability of our definitions to settings commonly considered in the literature, we first show how we can capture batch-job programs. To do so we define the set of possible results of a program as either a final memory state or \( \perp \) to indicate divergence.

As is common, assume a lattice of security levels \( (L, \sqsubseteq, \sqcup) \). An attacker at level \( \ell \) can see information at all levels \( \ell' \) as long as \( \ell' \sqsubseteq \ell \).

**Instantiation 1.** Let \( \mathbb{E} = \mathbb{M} \) where \( \mathbb{M} = \text{Var} \rightarrow \text{Val} \) for some set of variables \( \text{Var} \) and set of possible values \( \text{Val} \). We assume that there is a security level \( \Gamma(x) \in L \) associated with each variable \( x \in \text{Var} \). Let \( \mathbb{R} = \mathbb{M} \cup \{ \perp \} \).

This characterization then allows defining both termination-insensitive and termination-sensitive variants of noninterference by defining appropriate relations on \( \mathbb{R} \) and \( \mathbb{E} \).

We first define equivalence between initial memory states. Two memories are equivalent for the attacker if they coincide on all variables visible to the attacker:

**Definition 7.** \( m_1 =_\ell m_2 \iff \forall x. \Gamma(x) \sqsubseteq \ell \Rightarrow m_1(x) = m_2(x) \).

In the termination-insensitive [63] setting, termination is not observable to the attacker. Therefore, if either of the executions
diverges, the result is indistinguishable to the attacker. If both executions terminate, they must do so with equal values for observable variables.

**Definition 8** ($\sim_\ell$). $m_1 \sim_\ell m_2$ iff $(m_1 = \bot) \lor (m_2 = \bot) \lor (m_1 =_\ell m_2)$.

This allows defining termination-insensitive noninterference as follows:

**Definition 9** ($\text{TINI}(\ell)$). A program $c$ is termination-insensitively noninterferent for level $\ell$ (written $c \in \text{TINI}(\ell)$) iff $c \in \text{NI}(=\ell, \sim_\ell)$.

In the termination-sensitive [64] setting, the attacker is assumed to be able to observe whether or not a program terminates. Hence, two final memory states are indistinguishable to the attacker if and only if they either both diverge, or they coincide on variables visible to the attacker.

**Definition 10** ($\approx_\ell$). $m_1 \approx_\ell m_2$ iff $(m_1 = \bot \land m_2 = \bot) \lor (m_1 =_\ell m_2)$.

By instantiating $\sim_\ell$ in Definition 4 to $\approx_\ell$, we obtain termination-sensitive noninterference:

**Definition 11** ($\text{TSNI}(\ell)$). A program $c$ is termination-sensitively noninterferent for level $\ell$ (written $c \in \text{TSNI}(\ell)$) iff $c \in \text{NI}(=\ell, \approx_\ell)$.

To illustrate the difference between the two notions in the context of opacity, consider $\varphi_\bot = \{ m \mid m(h) \neq 1 \}$ and a program $c_\bot = \text{if } (h == 1) \{ \text{loop} \} \text{ else } \{ \text{skip} \}$. Since information is only leaked via termination, it holds that $\varphi_\bot \in \text{Op}(c_\bot, =_\ell, \sim_\ell)$ and $c_\bot \in \text{TINI}(\ell)$, but $\varphi_\bot \not\in \text{Op}(c_\bot, =_\ell, \approx_\ell)$ and $c_\bot \not\in \text{TSNI}(\ell)$.

**Noninterference and Opacity** Theorems connecting noninterference (without any declassification) and opacity can then be obtained by instantiating $\sim_\ell$ with $=_\ell$ in Lemma 4:

**Theorem 1** (Opacity and noninterference). Termination-(in)sensitive noninterference holds iff all informative predicates are termination-(in)sensitively opaque:

$$c \in \text{TINI}(\ell) \iff \forall M \in \ker(=\ell). \forall \varphi \subseteq M. \varphi \in \text{Op}(c, =_\ell, \sim_\ell)$$

$$c \in \text{TSNI}(\ell) \iff \forall M \in \ker(=\ell). \forall \varphi \subseteq M. \varphi \in \text{Op}(c, =_\ell, \approx_\ell)$$
1.4 I/O Programs

In an interactive setting, instead of assuming that the attacker can only see the final memory state of a program, they see a sequence of output events of the program. Inputs are modeled as memory states, a common simplification [2], similar to modeling environments as streams. Clark and Hunt [17] show that for the security of deterministic programs it makes no difference whether the environments are modeled as streams or strategies.

**Instantiation 2.** Let $E = M$ and $R = \mathbb{O}^*$ for some set of output events $\mathbb{O}$. We assume that each output event $o \in \mathbb{O}$ has an associated security level $\Gamma(o) \in \mathcal{L}$. We denote the empty trace by $\langle \rangle$ and appending trace $t$ to output event $o$ by $o.t$.

Using this type of outputs we can reason about both progress-insensitive [2, 3, 10], progress-sensitive [3], and timing-sensitive [56] notions of noninterference by suitably choosing the relation $\sim$ in $NI(\sim, \sim)$.

Progress-insensitive noninterference assumes that the attacker can not distinguish between the program silently diverging or the program producing more outputs in the future. Hence, for two traces to be progress-insensitively indistinguishable, they either have to coincide on low events or one of the traces has to diverge silently after producing a matching prefix of the other trace.

**Definition 12.** We define the projection $t \mid_\ell$ of trace $t$ to level $\ell$ recursively by $\langle \rangle \mid_\ell = \langle \rangle$ and $o.t \mid_\ell = \begin{cases} o.(t \mid_\ell) & \Gamma(o) \subseteq \ell \\ t \mid_\ell & \Gamma(o) \not\subseteq \ell \end{cases}$

**Definition 13.** $t_1 \sim_\ell t_2$ iff $(\exists t'_1, t''_1. t_1 \mid_\ell = t_2 \mid_\ell \land t_1 = t'_1.t''_1) \lor (\exists t'_2, t''_2. t_1 \mid_\ell = t'_2 \mid_\ell \land t_2 = t'_2.t''_2)$

This allows us to define progress-insensitive noninterference using the generic notion of noninterference introduced in Section 1.2.

**Definition 14 (PINI(\ell)).** A command $c$ is progress-insensitively noninterferent for level $\ell$ (written $c \in PINI(\ell)$) iff $c \in NI(=_\ell, \sim_\ell)$.

Progress-sensitive noninterference assumes that silent divergence is observable, so two traces have to match on all their low events:

**Definition 15 ($\approx_\ell$).** $t_1 \approx_\ell t_2$ iff $t_1 \mid_\ell = t_2 \mid_\ell$. 
As before, we define progress-sensitive noninterference by instantiating ∼ with ≈ℓ:

**Definition 16** (PSNI(ℓ)). A command c is progress-sensitively noninterferent for level ℓ (written c ∈ PSNI(ℓ)) iff c ∈ NI(=ℓ, ≈ℓ).

A further strengthening of this property can be obtained by assuming that the attacker also observes the occurrence of some high-security event but without knowing which exact event it is:

**Definition 17** (∼∼∼ℓ). t₁ ∼∼∼ℓ t₂ is defined inductively by the following rules

(1) ∅ ∼∼∼ ℓ \rightarrow o₁ = o₂ t₁ ∼∼∼ t₂

**Definition 18** (TimeNI(ℓ)). A command c is timing-sensitively noninterferent for level ℓ (written c ∈ TimeNI(ℓ)) iff c ∈ NI(=ℓ, ∼∼∼ℓ).

Time can be modeled using this definition by associating each computation with one or more tick events that model passing of time. In such a setting, computations where execution time depends on a secret, will not be noninterferent.

**Noninterference and Opacity** We can now use Lemma 4 to obtain a connection between the presented forms of noninterference and opacity:

**Theorem 2** (Opacity and noninterference). For ∼ ∈ {∼ℓ, ≈ℓ, ∼∼∼ℓ}, a program is noninterferent wrt. ∼ if all informative predicates are opaque for ∼:

\[ c \in PINI(ℓ) \leftrightarrow \forall M \in \ker(\sim). \forall \varphi \subseteq M. \varphi \in Op(c, =\ell, \sim) \]

\[ c \in PSNI(ℓ) \leftrightarrow \forall M \in \ker(\sim). \forall \varphi \subseteq M. \varphi \in Op(c, =\ell, \approx\ell) \]

\[ c \in TimeNI(ℓ) \leftrightarrow \forall M \in \ker(\sim). \forall \varphi \subseteq M. \varphi \in Op(c, =\ell, \approx\ell) \]

1.5 Information Release vs. Information Hiding

This section investigates the relation of opacity to the what dimension of declassification [58], which, as mentioned in Section 1.1, is most closely related when the focus is on protecting sensitive input.
As foreshadowed earlier, partial release [19, 41, 54, 55, 28, 43] policies specify what is released by splitting the domain of secrets into subdomains and only protecting secret variation within the subdomains. A convenient mechanism to specify partial release is via escape hatch expressions [55] that, intuitively, states that two initial environments are indistinguishable if and only if they agree on the values of the escape hatch expressions.

The accompanying policy of delimited release [55] specifies partial release by allowing to lower the security level of some expression, usually containing high variables, while prohibiting any other leaks beyond what is revealed by the declassified expression itself. Concretely, such policies can be expressed by adding expressions of the form declassify\((e)\) to the language in question.

Delimited release can be obtained from Definition 4 by suitably instantiating \(\sim\). To do so, we strengthen \(\approx\) by requiring that two memory states not only coincide on observable variables, but also on the values of declassified expressions.

**Definition 19** \((=^E_\ell)\). Two memories \(m_1, m_2\) are low equivalent for level \(\ell\) and set of declassified expressions \(E\) iff \(m_1 =_\ell m_2 \land (\forall e \in E. m_1(e) = m_2(e))\) where \(m(e)\) denotes evaluating expression \(e\) in memory \(m\).

**Definition 20** \((DR(\sim^o, \ell, E))\). A program \(c\) satisfies delimited release for level \(\ell\), relation \(\sim^o\), and declassified expressions \(E\) (written \(c \in DR(\sim^o, \ell, E)\)) iff \(c \in NI(=^E_\ell, \sim^o)\).

The intuition behind this definition is that resulting memory states only need to look the same to the attacker the declassified expressions also have equal values.

Lemma 4 also allows deriving results connecting delimited release and noninterference.

**Theorem 3** (Opacity and delimited release). For any reflexive relation \(\sim\), the following holds:

\[c \in DR(\sim^o, \ell, E) \iff \forall M \in \ker(=^E_\ell). \forall \varphi \subset M. \varphi \in Op(c, =^E_\ell, \sim^o)\]

In the batch setting this theorem covers both termination-sensitive and termination-insensitive noninterference. In the I/O setting, progress-sensitive, progress-insensitive, and timing-sensitive notions of noninterference are captured by the theorem statement.
1.6 Enforcement

1.6.1 Example Language

To illustrate our enforcement techniques, we introduce a simple \textit{while}-language with arrays and output. The command $\varepsilon$ denotes termination. Evaluation of a configuration $\langle c, m \rangle$ in one step to $\langle c', m' \rangle$ while producing trace $t$ is denoted by $\langle c, m \rangle \overset{t}{\rightarrow} \langle c', m' \rangle$. $\rightarrow^*$ denotes the reflexive-transitive closure of $\rightarrow$. We denote evaluation of an expression $e$ in memory $m$ by $m(e)$. Semantically, an array is modeled as a function mapping natural numbers to values. Program execution in this language is deterministic. The full definition and semantics of the language can be found in Appendix 1.A.1.

Without loss of generality, we consider only two security levels, $L$ for public data and $H$ for private data.

For the rest of this section, we instantiate the evaluation relation in the definitions of Section 1.2 as follows:

\textbf{Instantiation 3.} Let $\mathbb{R} = (\mathcal{L} \times \text{Val})^*$ and $\langle c, m \rangle \Downarrow t$ iff $\langle c, m \rangle \overset{t^*}{\rightarrow} \langle c', m' \rangle \land (\forall e'', m'', t'. \langle c', m' \rangle \overset{t^*}{\rightarrow} \langle c'', m'' \rangle \Rightarrow t' = \langle \rangle)$ for some $m'$.

1.6.2 Dynamic Monitoring

We present a dynamic enforcement mechanism inspired by common dynamic monitoring techniques for enforcing noninterference [24, 57, 35] and using concolic execution [59] for enforcing security properties [1].

The intuition is to keep track of the set of memories producing the same trace. Initially we consider the set $\{m_1\} \cap \varnothing$. This set becomes smaller over the course of execution, since outputs of the real execution need to be matched by another run. At each step of the program we check if the current set of possible starting memories is empty and, if so, stop execution before performing the next evaluation step.

To keep track of this, we define a function $\tau$ computing this set for the next step of a configuration along with keeping track of dependencies of variables, given starting memory $m_1$. $\tau(m_1, \langle c, m \rangle, M, \delta)$ produces a pair $(M', \delta')$ where $M'$ is a subset of memories in $M$ producing the same trace as $m_1$ for the next step that $\langle c, m \rangle$ takes.
In order to calculate which initial memories result in the same trace during an execution, we symbolically keep track of how variables change during the execution of a program. Concretely, we express the value of a variable at a certain point during execution in terms of variables in the initial state. For example, after executing two steps in the program $x := y; x := x * 2$, we record the value of $x$ as $y * 2$.

This is achieved by extending configurations with a function $\delta : \text{Var} \cup (\text{Var} \times \mathbb{N}) \rightarrow \mathbb{E} \times \mathcal{P}(\mathbb{E})$ that keeps track of variable dependencies in the following way: Evaluating the first component of $\delta(x)$ in $m_1$ yields the same result as $m_1'(x)$. The same holds for $m_2$ provided that $m_1$ and $m_2$ coincide on the second component of $\delta(x)$. The second component of $\delta(x)$ keeps track of which array operations have been performed during the execution that affected the value of $x$.

Similarly, $\delta$ keeps track of the dependencies of array elements $a[i]$. These notions are made precise by Lemma 7.

We extend $\delta$ to an arbitrary expression $e$ by replacing all variables and array lookups occurring in $e$ with their values in $\delta$. We denote this extension of $\delta$ applied to $e$ by $\delta(e,m)$. Since our approach is value-sensitive wrt. the array indices, the dependencies of an expression $e$ also depend on the current memory $m$.

**Definition 21.** We define $\delta(e,m) \in \mathbb{E} \times \mathcal{P}(\mathbb{E})$ for expression $e$ and memory $m$ recursively as follows:

$$
\begin{align*}
\delta(n,m) &= (n, \emptyset) \\
\delta(\text{true},m) &= (\text{true}, \emptyset) \\
\delta(\text{false},m) &= (\text{false}, \emptyset) \\
\delta(x,m) &= \delta(x) \\
\delta(a[e_i],m) &= (e', \{e'_i\} \cup E_i \cup E) \\
&\quad \text{where} \ (e', E) = \delta(a[m(e)]) \text{ and } (e'_i, E_i) = \delta(e_i, m) \\
\delta(e_1 \otimes e_2,m) &= (e'_1 \otimes e'_2, E_1 \cup E_2) \\
&\quad \text{where} \ (e'_1, E_1) = \delta(e_1,m) \text{ and } (e'_2, E_2) = \delta(e_2,m) \\
\delta((e_1,e_2),m) &= ((e'_1,e'_2), E_1 \cup E_2) \\
&\quad \text{where} \ (e'_1, E_1) = \delta(e_1,m) \text{ and } (e'_2, E_2) = \delta(e_2,m) \\
\delta(\text{fst}(e),m) &= (\text{fst}(e'), E) \quad \text{where} \ (e', E) = \delta(e,m) \\
\delta(\text{snd}(e),m) &= (\text{snd}(e'), E) \quad \text{where} \ (e', E) = \delta(e,m)
\end{align*}
$$
\[ \delta(\langle e, m \rangle) = (\langle e', E \rangle) \]

where \( (e', E) = \delta(\langle e, m \rangle) \)

\[ \delta(\langle e \ ? \ e_1 : e_2, m \rangle) = (\langle e' \ ? \ e'_1 : e'_2, E \cup E_1 \cup E_2 \rangle) \]

where \( (e', E) = \delta(\langle e, m \rangle) \), \( (e'_1, E_1) = \delta(\langle e_1, m \rangle) \) and \( (e'_2, E_2) = \delta(\langle e_2, m \rangle) \)

where \( \otimes \in \{+, -, \times, \leq, \geq, \&\& , ||\} \).

**Definition 22.** \( \tau \) is defined by:

\[ \tau(m_1, \langle \varepsilon, m \rangle, M, \delta) = (M, \delta) \]

\[ \tau(m_1, \langle \text{skip}, m \rangle, M, \delta) = (M, \delta) \]

\[ \tau(m_1, \langle x := e, m \rangle, M, \delta) = (M, \delta[x \mapsto \delta(\langle e, m \rangle)]) \]

\[ \tau(m_1, \langle a[e_1] := e_2, m \rangle, M, \delta) = (M \cap [m_1]_{\delta(e_1, m)}, \delta[a[m(e_1)] \mapsto \delta(\langle e_2, m \rangle)]) \]

\[ \tau(m_1, \langle \text{out} \ ? \ e, m \rangle, M, \delta) = \begin{cases} (M, \delta) & \ell' \nleq \ell \\ (M \cap [m_1]_{\delta(e, m)}, \delta) & \ell' \neq \ell \end{cases} \]

\[ \tau(m_1, \langle \text{if} \ e \{ \ c_1 \} \ \text{else} \{ \ c_2 \}, m \rangle, M, \delta) = (M \cap [m_1]_{\delta(e, m)}, \delta) \]

\[ \tau(m_1, \langle \text{while} \ e \ \text{do} \ c \rangle, M, \delta) = (M \cap [m_1]_{\delta(e, m)}, \delta) \]

\[ \tau(m_1, \langle c_1; c_2, m \rangle, M, \delta) = \tau(m_1, c_1, M, \delta) \]

where

\[ [m_1]_{\langle e, E \rangle} = \{ m_2 | m_1(e) = m_2(e) \land \forall e' \in E. m_1(e') = m_2(e') \} \]

For every case except assignments and array assignments, the mapping from variables to their dependencies is left unchanged. Producing an event on a public channel will constrain the set of possible starting memories to memories producing the same output for the expression.

Control-flow instructions, i.e. while and if statements are forced to take the same branches by allowing only memories in which the guard evaluates to the same value. Note that this still allows branching on low variables, as variables with low security levels
are required to be equal to $m_1$ in the set of considered memories anyway.

We then define the monitor via an instrumented semantics (for a fixed $m_1$) in Figure 1.2: As common for dynamic monitoring for information flow properties [24, 57], we do not inspect branches of conditionals or loops that are not taken during the run that is being monitored. Therefore, our definition of $\tau$ ensures that the set of possible starting memories not satisfying $\varphi$ takes the same branches as the run under consideration. Section 1.6.3 describes an approach that avoids this loss of precision at the cost of an increased performance overhead.

The following lemma makes the connection between the instrumented semantics and soundness precise:

**Lemma 7.** Whenever $\langle c, m_1, M, \delta_0 \rangle \xrightarrow{t_1}^* \langle c', m_1', M', \delta' \rangle$ and $m_2 \in M'$, then there exist $t_2, m_2'$ such that:

$\langle c, m_2 \rangle \xrightarrow{t_2}^* \langle c', m_2' \rangle \land t_1 \approx_L t_2$

and

$\forall e \in \mathcal{E}. m_1(\pi_1(\delta(e, m_1'))) = m_1'(e)$

and

$\forall e \in \mathcal{E}. m_1 = \pi_2(\delta(e, m_1')) = m_2 \Rightarrow$

$m_2(\pi_1(\delta(e, m_1'))) = m_2'(e)$

Figure 1.2: Instrumented semantics for monitoring
where \( \delta_0(x) = (x, \emptyset) \) and \( \delta_0(a[i]) = (a[i], \emptyset) \) and \( \pi_1 \) and \( \pi_2 \) denote the first and second projections of a tuple.

This allows establishing the soundness of the presented enforcement technique.

**Theorem 4** (Soundness of monitoring). If \( \langle c, m_1, [m_1] \circ \neg \varphi, \delta_0 \rangle \xrightarrow{t_1}^* \langle \varepsilon, m_1', M', \delta' \rangle \), then \( \varphi \in Op(c, =_L, \sim_L, m_1, t_1) \).

Similar to other dynamic enforcement mechanisms [57, 35], our soundness targets progress-insensitive security of monitored runs:

**Theorem 5.** If \( \langle c, m_1, [m_1] \circ \neg \varphi, \delta_0 \rangle \xrightarrow{t_1}^* \langle c_1', m_1', M', \delta' \rangle, \langle c, m_2 \rangle \xrightarrow{t_2}^* \langle c_2', m_2' \rangle, m_2 \in M', \) then \( t_1 \sim_L t_2, m_1 =_L m_2, \) and \( m_2 \notin \varphi \).

### 1.6.3 Sampling-based Enforcement

Being a fine-grained policy, opacity is not, in general, preserved by sequential composition of programs. In comparison, progress-sensitive noninterference is known to be compositional while progress-insensitive is not [51].

Consider predicate \( \varphi_{seq} = \{ m | m(h) \neq 5 \land m(h) \neq 6 \} \) and programs \( c_1 = (\text{if } (h == 5 \parallel h == 4) \{ \text{out L 1} \}; \text{else} \{ \text{out L 2} \}) \) and \( c_2 = (\text{if } (h == 6 \parallel h == 4) \{ \text{out L 3} \}; \text{else} \{ \text{out L 4} \}) \). Both \( c_1 \) and \( c_2 \) are opaque for \( \varphi_{seq} \): For \( c_1 \) and a memory \( m_1 \in \varphi_{seq} \), we can match the trace by a memory \( m_2 \notin \varphi_{seq} \) where \( m_2(h) = 6 \) if \( m_1(h) \neq 4 \) and \( m_2(h) = 5 \) otherwise. Analogously we can satisfy opacity for \( c_2 \). Moreover, notice that \( \varphi_{seq} \) also satisfies symmetric opacity for both \( c_1 \) and \( c_2 \).

However, neither \( \varphi_{seq} \) nor \( \overline{\varphi_{seq}} \) is opaque for the composition \( c_1; c_2 \): If \( m_1 \in \varphi_{seq} \) with \( m_1(h) \neq 4 \), then the trace \( t_1 = \langle 2, 4 \rangle \) is produced. In order for a memory \( m_2 \notin \varphi_{seq} \) to match the output \( 2, m_2(h) = 6 \) has to hold. For the output \( 4 \) to be matched, one needs to set \( m_2(h) = 5 \). Hence \( \varphi_{seq} \) is not opaque. Similarly if \( m_1 \notin \varphi \), matching the traces produced requires both \( m_2(h) = 4 \) and \( m_2(h) \neq 4 \).

The lack of compositionality motivates us to propose a blackbox randomized procedure to detect whether for a given initial memory \( m_1 \) satisfying a predicate \( \varphi \), there exists an equivalent run starting in a memory \( m_2 \notin \varphi \). A high-level description of the algorithm is displayed below. This is a blackbox approach because the program
code is not inspected, but certain outputs in the trace are used for
the heuristics for the sampling process, detailed in Section 1.7.

**Input:** Program $c$, predicate $\varphi$, initial memory $m_1$, where $m_1 \in \varphi$.

**Parameter:** Sampling function $S : C \times M \to \mathcal{P}([m_1]_\ell)$. We assume that for all $c$ and $m_1$ that $S(c, m_1)$ is finite.

**Output:** Memory $m_2$ and $t_2$ satisfying $\langle c, m_2 \rangle \downarrow t_2$, $m_1 \sim m_2$, $t_1 \sim t_2$, and $m_2 \not\in \varphi$.

\[
\text{for } m_2 \in S(c, m_1) \text{ do }
\begin{align*}
\quad & \text{if } [[\langle c, m_1 \rangle]] \sim [[\langle c, m_2 \rangle]] \land m_2 \not\in \varphi \text{ then } \\
\quad & \quad \text{return } (m_2, [[\langle c, m_2 \rangle]]) \\
\quad \text{end}
\end{align*}
\]

Where $[[\langle c, m \rangle]]$ denotes the trace produced by executing $c$ starting in memory $m$. Note that this trace is uniquely defined since the language is deterministic.

The advantage of this approach over the dynamic monitoring technique described in Section 1.6.2 is that it does not force all runs that are being considered to take the same branches for if-statements and execute while-loops the same number of times. However, this incurs a performance overhead as the program is executed multiple times. In our implementation, $S$ is constructed using the random testing tool QuickCheck [16], as detailed in Section 1.7.

The following theorem establishes the soundness of this approach:

**Theorem 6** (Soundness of Sampling-based Enforcement). If the above algorithm returns a pair $(m_2, t_2)$, for initial memory $m_1$ resulting in trace $t_1$, then $\varphi \in Op(c, =_\ell, =_\omega, m_1, t_1)$.

1.7 Experiments

To demonstrate the practicality of the enforcement, we implement both the monitoring sampling-based approaches from Sections 1.6.2 and 1.6.3 in Haskell using BNFC [9] for parser generation. The source code is also available online.
To provide more realistic examples we add a facility to generate random numbers to the example language presented in Section 1.6.1. Statement \texttt{randomize}(x) assigns a random number to variable \texttt{x}. Note that this can be emulated by instead computing the outputs of a deterministic pseudo-random number generator programmatically, based on a private variable \( h_{\text{seed}} \) with \( \Gamma(h_{\text{seed}}) \not\subseteq \ell \), which is then updated after each use of the \texttt{randomize}(x) statement. Since we assume randomness to be unpredictable, we require that \( h_{\text{seed}} \) not occur in predicates.

For simplicity, we specify predicates as expressions in the language from Section 1.6.1, sufficient to express all predicates in the examples. The enforcement approaches are applicable to more complex languages for expressing predicates.

### 1.7.1 Location Privacy

We apply our enforcement to enforcing location privacy code (with the exception of Program 1, which is subsumed by Program 2).

**Dynamic Monitoring.**

To implement the sets of possible memories that are possible at each point, we collect constraints created by execution steps of a program and the predicate. We utilize state-of-the-art SMT solvers (Z3 [20] and CVC4 [7]) to ensure that the set of constraints is satisfiable, thereby showing that the set of memories producing the same trace is nonempty. We use multiple SMT solvers to cover cases where one solver might not be able to solve a particular problem. In the cases we examined however, both solvers were able to handle the generated formulas with Z3 often being faster.

As common for purely dynamic enforcement [57, 35] we do not inspect the branches not taken, which makes a difference for Program 2 with the clinic scenario for the monitoring and sampling techniques. We will see that the program is susceptible to sampling-based enforcement while monitoring takes the same branches of \texttt{if} statements in the set of memories considered and hence has insufficient information to verify the program.

However, we can make this program amenable to dynamic enforcement by unconditionally computing random coordinates and
then deciding which set of coordinates to output in the expression itself, as shown in Program 2a.

```c
/* Program 2a - Adapted */
/* Location privacy with random output */
clinicXmin := 200; clinicXmax := 400;
clinicYmin := 50; clinicYmax := 150;
randomize(x);
while (x >= clinicXmin && x <= clinicXmax) {
    randomize(x);
}
randomize(y);
while (y >= clinicYmin && y <= clinicYmax) {
    randomize(y);
}
out L ((hX >= clinicXmin && hX <= clinicXmax &&
        hY >= clinicYmin && hY <= clinicYmax)
    ? (x, y)
    : (hX, hY));
```

Note that this example also eliminates a possible timing leak introduced by performing the random number generation only if the user is located inside the clinic.

We introduce variables for the SMT solver for all variables occurring in the program and the predicate. The confidential information is whether the user is currently in the medical clinic, i.e. $\varphi_{\text{loc}} = \{m | m(hX) \geq \text{clinicXmin} \land m(hX) \leq \text{clinicXmax} \land m(hY) \geq \text{clinicYmin} \land m(hY) \leq \text{clinicYmax}\}$, where $\text{clinicX}_{\text{min, max}}$ refer to the start and end of the clinic location on the X-axis and $\text{clinicY}_{\text{min, max}}$ refer to the extent of the clinic location on the Y-axis.

If $\varphi_{\text{loc}}$ is satisfied in the initial memory $m_1$, then the user is in the clinic. This fact should not be disclosed, and hence we need to find a memory $m_2$ such that $m_2 \notin \varphi_{\text{loc}}$. Therefore, we add $-(m(hX) \geq \text{clinicX}_{\text{min}} \land m(hX) \leq \text{clinicX}_{\text{max}} \land m(hY) \geq \text{clinicY}_{\text{min}} \land m(hY) \leq \text{clinicY}_{\text{max}})$ to the set of constraints (we assume that $\varphi_{\text{loc}}$ holds in $m_1$, but the enforcement approach works in the same way for symmetric opacity).

Moreover, $m_2$ needs to coincide with $m_1$ on low variables, hence we require all low variables occurring in the program to be equal to their values in $m_1$. In this case, no low variable is used before being overwritten, so these constraints are vacuous. For brevity, we omit them here.

During evaluation of assignment statements the mapping $\delta$ is
then updated with the dependencies of the variables. For a \texttt{randomize}(x) instruction, $\delta(x)$ is updated with a fresh random variable since the result of the random number generation is assumed to be unpredictable. In the case of an assignment $x := e$ we set the dependencies of $x$ to $\delta(e,m)$.

When encountering a \texttt{while}-loop with expression $e$ as the guard, we then add the constraint that the initial memory must agree with $m_1$ on $\delta(e,m)$. In particular, we generate constraints stating the guards on both loops coincide with $m_1$, i.e. that the random variables introduced for calls to \texttt{randomize()} result in the same number of loop iterations.

When evaluating the \texttt{out} statement we then add the constraint that the output of the run starting with $m_1$ has to be matched by a run starting in a memory $m_2$ satisfying the generated constraints.

In this case the run of the program starting in $m_1$ will result in the output $(x,y)$, where $x$ and $y$ are randomly generated, but outside the clinic. For example, assume that $(x,y)$ evaluates to $(23,45)$ for $m_1$. Therefore we add the constraint $(23,45) = (hX \geq 100 \land hX \leq 200 \ldots ? (r_1, r_2) : (hX, hY)$ where $r_1$ and $r_2$ are the fresh variables introduced by the \texttt{randomize()} instructions.

Whenever a constraint is added during monitoring, we translate the constraints into the syntax of the SMT solvers being used and check for satisfiability. If the solver returns that the set is satisfiable, we have found a memory producing the same trace which does not satisfy $\varphi_{\text{loc}}$ and hence no information about $\varphi_{\text{loc}}$ is revealed. In this particular case, the SMT solver will find a memory $m_2$ such that $m_2(hX) = 23$ and $m_2(hY) = 45$. $r_1$ and $r_2$ can have any coordinates outside the clinic.

Generally, we gain from SMT solvers to help with satisfying the constraints, with the tradeoff that when the set of constraints is unsatisfiable or the solver times out, then we would block the execution to prevent a possible opacity violation.

**Sampling-based Enforcement.**

Sampling-based enforcement consists of two steps: Running program $c$ with the initial memory $m_1$ and trying random memories $m_2$ (with $m_1 \sim m_2$) to check if they produce the same trace where $\varphi$ no longer holds. We use the QuickCheck tool [16] to generate
random samples.

To handle randomness in our language, we add the following heuristic to our sampling-mechanism: If, during evaluation with memory \( m_1 \), we output values \( v_1, \ldots, v_n \), we increase the likelihood of choosing \( v_1, \ldots, v_n \) for variables \( x \in \text{vars}(c) \) where \( \Gamma(x) = H \).

This heuristic allows us to check both the original clinic code in Program 2 (with if-statements to decide what to output), as well as the one adapted to dynamic monitoring.

Assuming that we start with a memory \( m_1 \) where \( (m_1(hX), m_2(hY)) \) is located within the medical clinic, we will output random coordinates \( (r_1, r_2) \) outside of the clinic. Using the heuristic described above, our sampling mechanism will consider memories \( m_2 \) with \( m_2(hX) \in \{r_1, r_2\} \) and \( m_2(hY) \in \{r_1, r_2\} \) with increased likelihood.

Hence, the presented approach finds a witness for opacity for this example after only a few tested memories.

**Progress-sensitivity**

Consider the example presented in Program 3 which does not produce any output if the user is located in a sensitive location. If the user is located outside of a sensitive area, their real coordinates are output.

This example satisfies progress-\textit{insensitive} opacity, since the empty trace (produced if the user is in a sensitive location) is a prefix of all other traces.

However, progress-\textit{sensitive} opacity is violated since the low output event that is generated if the user is located outside of sensitive areas cannot be matched in a run starting in a sensitive location.

Since this example relies on branching to achieve opacity, dynamic monitoring cannot be used to run this program. However, sampling-based enforcement is able to verify that the example satisfies progress-insensitive opacity, while progress-sensitive opacity correctly cannot be established.

### 1.7.2 Statistics Aggregation

We apply our enforcement to the healthcare statistics code in Program 4.
Dynamic Monitoring.

Assume a program run with initial memory $m_1$ results in 7. Assume the sensitive predicate $\varphi = \{m_1 \mid m_1(h\text{HasDisease}[3]) > 0\}$, i.e. whether the fourth patient is infected with a particular disease. Moreover, assume that $m_1(h\text{HasDisease}[3]) = 1$, i.e. that $m_1 \in \varphi$.

The monitor will initially add the constraints that $\varphi$ must not be satisfied and that all low variables have the same values as in $m_1$. Among the constraints generated, when reaching the statement \texttt{out L result}, we add:

\[
7 = 0 + (h\text{HasDisease}[0] > 0 \ ? \ 1 : 0) \\
+ \cdots + (h\text{HasDisease}[9] > 0 \ ? \ 1 : 0)
\]

Running an SMT solver on the set of generated constraints yields that they are satisfiable: Since the sum of ten values of $m_1$ is 7, $m_1(h\text{HasDisease}[i]) = 0$ must hold for some $i \neq 3$. Therefore, we can satisfy the set of constraints by setting $m_2(h\text{HasDisease}[3]) = 0$ and $m_2(h\text{HasDisease}[i]) = i$ and $m_2(x) = m_1(x)$ otherwise. This memory yields the same observations and $m_2 \notin \varphi$ does not hold.

Consider now instead a run with memory $m'_1$ resulting in the output 10. In this case, the monitor will end up with an unsatisfiable set of constraints, since this implies that all patients are infected with the disease. Since the total number of patients is known to the attacker, they can infer that the fourth patient must also have this disease. Hence, this run is correctly terminated by our enforcement before the output takes place.

Sampling-based Enforcement.

As before, the sampling-based technique quickly finds a suitable witness for opacity due to heuristics employed by the \textit{QuickCheck} library.

1.7.3 Discussion

The presented monitoring and sampling mechanisms offer different tradeoffs of precision and performance.

Monitoring avoids executing the program multiple times and utilizes SMT solving for problems that tend to be small, relative to the capabilities of SMT solvers. Moreover, the satisfiability only
needs to be verified when encountering \textbf{out} expressions. On the other hand, programs are not allowed to branch on secrets, reducing precision. In some cases, these programs can be amended to allow for execution with the monitor, e.g. for Program 2.

Being blackbox, sampling-based enforcement is scalable to rich languages. While sampling-based enforcement is more precise, it uses heuristics for choosing an appropriate starting environment that leads to a higher rate of successful tries.

The table below summarizes the results for the two enforcement mechanisms concerning the examples from Section 1.1. The checkmarks represent successful verification with respect to the provided indistinguishability relations.

<table>
<thead>
<tr>
<th>Example</th>
<th>Sampling</th>
<th>Monitoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 2</td>
<td>✓ (≈)</td>
<td>✓ (≈)</td>
</tr>
<tr>
<td>Program 2a</td>
<td>✓ (≈)</td>
<td>✓ (≈)</td>
</tr>
<tr>
<td>Program 3</td>
<td>✓ (∼)</td>
<td>✓ (≈)</td>
</tr>
<tr>
<td>Program 4</td>
<td>✓ (≈)</td>
<td>✓ (≈)</td>
</tr>
</tbody>
</table>

As a final note, our implementation is a proof-of-concept implementation, with a number of possible performance optimizations such as reducing the number of spawned processes when handling constraints. While these optimizations and scalability studies are promising directions of future work, we note that indicative performance overhead does not strike as unacceptable: sampling-based enforcement runs in a few milliseconds and dynamic monitoring within a few hundred milliseconds for each of the examples (run on an Intel i7-4600 processor using Linux 3.15.2 and compiled with GHC 7.8.3). While not insignificant, the proof-of-concept prototype can be a good fit for testing applications before deployment.

### 1.8 Related Work

The origins of opacity can be traced back to Sutherland’s \textit{nondeducibility} [60], with the intuition of keeping attacker-observable events consistent with possible variations of secret inputs. Nondeducibility has been criticized for failing to protect secret outputs [34] and address covert channels [66]. These criticisms have no bearing on the stream-based setting as in this paper, but need
1.8. Related Work


Hughes and Shmatikov [39] develop an algebraic theory of *opacity* for reasoning about general knowledge functions. They present a protocol graph framework and study a hierarchy of anonymity system properties, noting that anonymity is neither necessary nor sufficient for privacy.

Bryans et al. study opacity in the setting of Petri nets [14] and in a general setting of transition systems [12, 13].

Ryan and Peacock [53] explore the relation between noninterference, noninference [50], nondeducibility [60], and nonleakage [65] in the setting of labeled transition systems expressed in CSP [37]. Although cast in a different setting and leaving out support for declassification and enforcement, the connection between opacity and noninterference is particularly relevant. They state that “formulating noninterference as opacity proves difficult, but we can show that noninterference implies opacity”. Our work takes the next steps by connecting noninterference and opacity in both directions, parameterizing the results in the power of the attacker and declassification policies, and developing enforcement mechanisms.

Freni et al. [27] treat location privacy in social networks. Of particular interest is the definition of *absence privacy* to protect the fact that a user is absent at certain location points (reducing burglary risks). However, absence privacy requires for all locations $p$ in sensitive regions that the adversary cannot exclude that the user is located in $p$. With opacity, there is no need to demand excluding all locations: with home as a sensitive area, the fact that the user is absent in the kitchen is not dangerous when the user is present in the living room.

Although probabilistic behavior is not in the scope of our model, a worthwhile direction for further investigations is to extend it with probabilities. It would be interesting to combine it with the work by Bérard et al. [8] who study several probabilistic opacity properties in the setting of probabilistic automata.

In the context of databases, properties similar to opacity are desired for the *inference problem* [21] concerned with protecting individual data while revealing statistical aggregates. Mechanisms such as data swapping and query size control have been developed...
and their limitations identified. It is often possible to subvert these mechanisms by correlating the query results [23].

In an investigation of provenance security, Cheney [15] defines the provenance obfuscation problem for protecting database queries. The problem consists of inability of users to answers queries using their observations. A query cannot be answered if for any trace, there exists another trace with the same observation but so that the results of applying the query to the traces are different. Intuitively, this in line with symmetric opacity, although the exact relation is yet to be established by future work.

Del Tedesco et al. [61] study logical data erasure and develop a semantic hierarchy of erasure policies. When exploring the choices for ordering between knowledge sets that result from observing system behavior, they utilize a notion akin to opacity for modeling facts and queries over knowledge sets. While the main focus is on the expressiveness of erasure policies, verification of erasure policies is left for future work.

Griffis et al. [31] focus on the problem of personal data vaults to separate the capturing and sharing of data. In similar spirit to ours, they argue that common security definitions fall short of capturing location sharing policies whose granularity depends on the actual location. They propose to use filters to augment information release mechanisms by transforming sensitive information into coarse-grained approximations.

Gruska reasons about passive and active timing attacks in the context of opacity for timed process algebra [32] and explores opacity for both confidentiality and integrity [33] in a variant of CCS [47].

Hritcu et al. [38] utilize QuickCheck to aid the process of proving noninterference for a low-level abstract information-flow machine. The approach is directly suitable for checking unwinding conditions [29].

Wu and Lafortune [67] investigate a family of opacity policies for deterministic finite-state automata. They propose verification methods that are suitable for certifying opacity in the presence of a team of collaborating intruders.

The line of work on abstract noninterference [28, 40, 45] leads up to a powerful generalization by Mastroeni [45], formulated in a noninterference style with quantifying over pairs of runs with initial states indistinguishable by the attacker. The generalization
is parametric in the indistinguishability on both inputs and outputs, which can be used for hiding differences between inputs.

Relation to declassification policies deserves discussion in the rest of the section. Let us come back to Program 1 where the goal is to make opaque whether the user is inside the hospital. Traditional declassification mechanisms require specifying what is released when the user is located in the clinic, i.e. the method of masking the user’s location in that case is part of the policy. For example, declassification via escape-hatch expressions would allow expressing the policy in this scenario by downgrading the information using an expression of the form \texttt{declassify}(hX \geq clinicXmin \ldots ? (100, 200) : (hX, hY)). From a policy standpoint however, it is unimportant how exactly the user’s real location is concealed if he is located in a sensitive area, making opacity a more natural fit for this scenario.

More elaborate approaches for specifying conditional declassification policies have been proposed. Banerjee et al. [6] present an approach allowing to specify under which condition confidential information can be released while still providing delimited-release style guarantees that nothing else is leaked, using \texttt{flowspecs}. Nanevski et al. [49] propose a framework for expressing information flow policies based on dependent types. Their approach allows constructing functions that declassify information only under specified conditions.

Opacity policies however remain non-trivial to express in such frameworks. Fundamentally, the above frameworks allow expressing under which conditions output that is released to the attacker has to be equal in two runs that vary in the parts of secret input that may be released. In the case of Program 1, however, whenever the user is located outside of the clinic, the outputs of the program need not be equal between the two run. Moreover, the crucial comparison for opacity is between runs that differ on whether or not the user is located in the clinic; i.e. when the criterion for declassification is true in one run and violated in the other. Such policies are not natural to express under conditional release.

Generally, declassification deals with what \textit{values} can be released, whereas opacity is concerned with \textit{properties}. If properties are to be protected, it can be hard to see whether or not releasing a value will affect the attacker’s knowledge about whether or not this
property holds, especially if the property involves several variables.

1.9 Conclusions

Driven by the research questions on understanding and enforcing opacity, we have presented a formal framework for opacity and demonstrated its differences and similarities to the common security definitions of noninterference, knowledge-based security, and information release. Our results give insight into the formal relation to the common policies, which can be achieved by quantifying over the system properties that need to be opaque. These results are parametric in the power of the attacker and formalized in Isabelle/HOL.

Our policy framework is accompanied by two enforcement strategies: a whitebox monitor and a blackbox sampling-based enforcement. We have established the soundness of the mechanisms and showed their usefulness by a prototype for the scenarios of location privacy and privacy-preserving aggregation.

Future work

A promising track for future work is exploring epistemic logic for expressing and enforcing opacity as well as applying it to reason about surreptitious code and program obfuscation [48]. The recent work on epistemic logic for information-flow security [5, 18] is a promising starting point.

In general, which properties should be opaque depends on the application and what properties of user input are desired to be protected. Weaker or stronger predicates might be appropriate, with no systematic way available a priori. Deriving opacity properties from code, to guide policy makers in tuning their policies, is an intriguing problem to investigate.

Our location privacy policies exemplify typical static policies, as commonly used in location privacy protocols [42, 62], and as used in Flickr’s geofences. These policies need to be refined when disclosing location over time. For example, if a patient on the way to a hospital exposes a trajectory leading up to a geofence that surrounds the hospital, the attacker might conclude that the patient is at the hospital at a later time. Tracking location over
time is a major challenge for much work on location privacy [42, 62]. Developing a sharper analysis that incorporates topological, spatial, and temporal sensitivity is much desired. We believe that opacity can be fruitfully applied to describe properties on trajectories

Acknowledgments

This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

Bibliography


Chapter 1: Opacity


1.A Appendices

1.A.1 Language Semantics

Figure 1.3 shows the syntax of expressions (e), commands (c), and values (V). We assume that each array a is declared in advance. Output is produced by instructions of the form out ℓ e where ℓ is the level of the resulting event.

For simplicity, we make various well-formedness assumptions on programs:

- All programs are well-typed, i.e. expressions like 1 + true are not considered. Similarly, guards of conditionals are boolean expressions.
- Array variables and non-array variables are disjoint.
\[ \ell ::= L | H \]

\[ e ::= x | n | \text{true} | \text{false} \]
\[ | e + e | e \times e | e - e \]
\[ | e \&\& e | e || e | !e | e ? e : e \]
\[ | e \leq e | e == e | e \geq e \]
\[ | (e, e) | \text{fst}(e) | \text{snd}(e) | a[e] \]

\[ c ::= \varepsilon | \text{skip} | x ::= e | \text{out} \ell e | c ; c | \text{if } e \{ c \} \text{ else } \{ c \} | \text{while } e \text{ do } c | a[e] ::= e \]

\[ V ::= n | \text{true} | \text{false} | (V, V) \]

Figure 1.3: Syntax of example language

We assume that memories map variables to values and arrays to functions mapping natural numbers to values.

These assumptions are specified precisely in the Isabelle/HOL formalization.

**Definition 23** (Expression evaluation). *Evaluation an expression* \( e \) *in memory* \( m \), denoted by \( m(e) : V \) *is defined recursively as follows:*

\[
\begin{align*}
m(n) &= n \\
m(\text{true}) &= \text{true} \\
m(\text{false}) &= \text{false} \\
m(x) &= m(x) \\
m(e_1 + e_2) &= m(e_1) + m(e_2) \\
m(e_1 - e_2) &= m(e_1) - m(e_2) \\
m(e_1 \times e_2) &= m(e_1) \times m(e_2) \\
m((e_1, e_2)) &= (m(e_1), m(e_2)) \\
m(\text{fst}(e)) &= \begin{cases} v_1 & m(e) = (v_1, v_2) \\ v_2 & m(e) = (v_1, v_2) \end{cases} \\
m(\text{snd}(e)) &= \begin{cases} v_2 & m(e) = (v_1, v_2) \\ \text{true} & m(e_1) \leq m(e_2) \end{cases} \\
m(e_1 \leq e_2) &= \begin{cases} \text{true} & m(e_1) \leq m(e_2) \\ \text{false} & \text{otherwise} \end{cases}
\end{align*}
\]
Chapter 1: Opacity

\[
m(e_1 \geq e_2) = \begin{cases} 
    \text{true} & m(e_1) \geq m(e_2) \\
    \text{false} & \text{otherwise}
\end{cases}
\]

\[
m(e_1 \&\& e_2) = \begin{cases} 
    \text{true} & m(e_1) = \text{true} \land m(e_2) = \text{true} \\
    \text{false} & \text{otherwise}
\end{cases}
\]

\[
m(e_1 \parallel e_2) = \begin{cases} 
    \text{true} & m(e_1) = \text{true} \lor m(e_2) = \text{true} \\
    \text{false} & \text{otherwise}
\end{cases}
\]

\[
m(!e_1) = \begin{cases} 
    \text{true} & m(e_1) = \text{false} \\
    \text{false} & \text{otherwise}
\end{cases}
\]

\[
m(e_1 == e_2) = \begin{cases} 
    \text{true} & m(e_1) = m(e_2) \\
    \text{false} & \text{otherwise}
\end{cases}
\]

\[
m(e \ ? e_1 : e_2) = \begin{cases} 
    m(e_1) & m(e) = \text{true} \\
    m(e_2) & \text{otherwise}
\end{cases}
\]

\[
m(a[e]) = m(a)(m(e))
\]

Figure 1.4 shows the small-step semantics where \(\langle c, m \rangle \xrightarrow{t} \langle c', m' \rangle\) denotes evaluation of \(\langle c, m \rangle\) to \(\langle c', m' \rangle\) in one step, producing trace \(t\). We omit \(t\) if it is empty.

**Definition 24.** Let \(\rightarrow^*\) be the reflexive-transitive closure of \(\rightarrow\), i.e. \(\langle c, m \rangle \not\rightarrow^* \langle c, m \rangle\) and \(\langle c, m \rangle \xrightarrow{t_1} \langle c', m' \rangle\) if \(\langle c, m \rangle \xrightarrow{t_1}^* \langle c'', m'' \rangle\) and \(\langle c'', m'' \rangle \xrightarrow{t_2}^* \langle c', m' \rangle\) for some \(c'', m''\).

1.A.2 Connecting Noninterference and Opacity

1.A.3 Proofs

Framework

*Proof of Lemma 1.* Clear from expanding the definition of \(Op(c, \overset{i}{\sim}, \overset{o}{\sim})\). \(\square\)

*Proof of Lemma 2.* Let \(\varphi_1 \in Op(c, \overset{i}{\sim}, \overset{o}{\sim}) \lor \varphi_2 \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})\), \(\langle c, e_1 \rangle \downarrow r_1\), and \(e_1 \in \varphi_1 \cap \varphi_2\). Hence we have that \(e_1 \in \varphi_1\) and \(e_2 \in \varphi_2\). We have \(\varphi_i \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})\) for some \(i \in \{1, 2\}\). Hence there
exist $e_2, r_2$ such that $\langle c, e_2 \rangle \Downarrow r_2, e_1 \overset{i}{\sim} e_2, r_1 \overset{o}{\sim} r_2$ and $e_2 \notin \varphi_i$. The latter also implies $e_2 \notin \varphi_1 \cap \varphi_2$, which shows $\varphi_1 \cap \varphi_2 \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$. \hfill \square
Proof. Let \( \varphi_1 \cup \varphi_2 \in Op(c, \hat{\sim}, \tilde{\sim}) \), \( \langle c, e_1 \rangle \Downarrow r_1 \), and \( e_1 \in \varphi_i \) (for \( i = 1, 2 \)). Hence we also have \( e_1 \in \varphi_1 \cup \varphi_2 \). By the opacity assumption there then exist \( e_2 \) and \( r_2 \) such that \( \langle c, e_2 \rangle \Downarrow r_2 \), \( e_1 \hat{\sim} e_2 \), \( r_1 \hat{\sim} r_2 \) and \( e_2 \not\in \varphi_1 \cup \varphi_2 \). Hence we also have \( e_2 \not\in \varphi_i \), allowing us to conclude \( \varphi_i \in Op(c, \hat{\sim}, \tilde{\sim}) \) (for \( i = 1, 2 \)).

Proof of Lemma 4. \( \Rightarrow \): Let \( c \in NI(\hat{\sim}, \tilde{\sim}) \) and assume \( E \in \ker(\hat{\sim}) \) and \( \varphi \subseteq E \). We need to show that \( \varphi \in Op(c, \hat{\sim}, \tilde{\sim}) \).

For that, assume \( \langle c, e_1 \rangle \Downarrow r_1 \) and \( e_1 \in \varphi \). Since \( \varphi \subseteq M \) there exists an environment \( e_2 \) such that \( e_2 \in E \land e_2 \not\in \varphi \). Since \( e_2 \in \varphi \) and \( \varphi \subseteq E \), it follows that \( e_1 \hat{\sim} e_2 \), by definition of \( \ker(\hat{\sim}) \). Since \( c \in NI(\hat{\sim}, \tilde{\sim}) \), there then must exist a result \( r_2 \) such that \( \langle c, e_2 \rangle \Downarrow r_2 \) and \( r_1 \hat{\sim} r_2 \) as desired.

\( \Leftarrow \): Assume \( \forall E \in \ker(\hat{\sim}) \forall \varphi \subseteq M. \varphi \in Op(c, \hat{\sim}, \tilde{\sim}) \) and let \( \langle c, e_1 \rangle \Downarrow r_1 \) and \( e_1 \hat{\sim} e_2 \). We need to exhibit a result \( r_2 \) such that \( \langle c, e_2 \rangle \Downarrow r_2 \) and \( r_1 \hat{\sim} r_2 \).

We show our goal by first excluding the trivial case where \( e_2 \) is equal to \( e_1 \), so we then proceed by case distinction on \( e_1 = e_2 \):

Case \( e_1 = e_2 \): In this case, we can satisfy noninterference by setting \( r_2 = r_1 \). The conclusion follows since \( \hat{\sim} \) is assumed to be reflexive. Case \( e_1 \neq e_2 \): Since \( e_1 \hat{\sim} e_2 \), we have that \( e_1, e_2 \in [e_1]_{\hat{\sim}} \), with \( [e_1]_{\hat{\sim}} \in \ker(\hat{\sim}) \).

In this case we define a predicate \( \varphi = [e_1]_{\hat{\sim}} \setminus \{e_2\} \). Since \( e_2 \not\in \varphi \), but \( e_2 \in [e_1]_{\hat{\sim}} \), we have \( \varphi \subseteq [e_1]_{\hat{\sim}} \). By assumption it then holds that \( \varphi \in Op(c, \hat{\sim}, \tilde{\sim}) \).

By definition of opacity, there must then exist an environment \( \tilde{e}_2 \) and a result \( \tilde{r}_2 \) such that:

\[
\langle c, \tilde{e}_2 \rangle \Downarrow \tilde{r}_2 \land e_1 \hat{\sim} \tilde{e}_2 \land r_1 \hat{\sim} \tilde{r}_2 \land \tilde{e}_2 \not\in \varphi
\]

In this case, by construction of \( \varphi \), we have from \( \tilde{e}_2 \not\in \varphi \) that \( \tilde{e}_2 = e_2 \). Noninterference can therefore be satisfied by setting \( r_2 = \tilde{r}_2 \).

Proof of Lemma 5. \( \Leftarrow \): Clear, since for each equivalence class \( E \) and set \( E \setminus \{e\} \) for \( e \in E \), it holds that \( (E \setminus \{e\}) \not\subseteq E \).
⇒: Assume ∀E ∈ ker(\(\tilde{\sim}\)). ∀e ∈ E. (E \ {e}) ∈ Op(c, \(\tilde{\sim}\), \(\sim\)), E ∈ ker(\(\tilde{\sim}\)), and \(\varphi \subseteq E\). We have to show that \(\varphi \in Op(c, \(\tilde{\sim}\), \(\sim\))\).

For that, let \(\langle c, e_1 \rangle \Downarrow r_1\) and \(e_1 \in \varphi\). Since \(\varphi \subseteq E\), there is an environment \(e_2 \in E\) such that \(e_2 \not\in \varphi\). Consider the set \(E \setminus \{e_2\}\).

By assumption, this set is opaque for \(c, \(\tilde{\sim}\), and \(\sim\). Hence there exists an environment \(\tilde{e}_2\) and a result \(r_2\) such that \(\langle c, \tilde{e}_2 \rangle \Downarrow r_2 \wedge e_1 \overset{i}{\sim} \tilde{e}_2 \wedge r_1 \overset{o}{\sim} r_2 \wedge \tilde{e}_2 \not\in (E \setminus \{e_2\})\).

From \(\tilde{e}_2 \not\in (E \setminus \{e_2\})\) it follows that \(\tilde{e}_2 = e_2\). Hence \(\langle c, e_2 \rangle \Downarrow r_2 \wedge e_1 \overset{i}{\sim} e_2 \wedge r_1 \overset{o}{\sim} r_2 \wedge e_2 \not\in \varphi\) as desired.

\[\begin{align*}
\text{Proof of Lemma 6. } &\Rightarrow: \text{ Assume } \varphi \in Op(c, \(\tilde{i}\), \(\tilde{o}\)), e_0 \in E, \text{ and } r_0 \in \mathbb{R}(c, e_0). \text{ Therefore, it holds that } \langle c, e_0 \rangle \Downarrow r_0 \text{ by definition of } \mathbb{R}(c, e_0).

\text{Since } \varphi \text{ is opaque, there exist } e_2, r_2 \text{ such that } \langle c, e_2 \rangle \Downarrow r_2 \wedge e_0 \overset{i}{\sim} e_2 \wedge r_0 \overset{o}{\sim} r_2 \wedge e_2 \not\in \varphi. \text{ By definition of } k, \text{ we then have that } e_2 \in k(c, \(\tilde{i}\), \(\tilde{o}\), e_0, r_0). \text{ Moreover, since } e_2 \not\in \varphi, \text{ it also holds that } e_2 \in \varphi. \text{ Hence, } k(c, \(\tilde{i}\), \(\tilde{o}\), e_0, r_0) \cap \varphi \text{ is nonempty, as desired.}

\text{\(\Leftarrow\): Assume } \forall e_0. \forall r_0 \in \mathbb{R}(c, e_0).k(c, \(\tilde{i}\), \(\tilde{o}\), e_0, r_0) \cap \varphi \neq \emptyset. \text{ Moreover, assume } \langle c, e_1 \rangle \Downarrow r_1. \text{ Hence we have } r_1 \in \mathbb{R}(c, e_1) \text{ and therefore } \\
k(c, \(\tilde{i}\), \(\tilde{o}\), e_1, r_1) \cap \varphi \neq \emptyset.

\text{Therefore, there exists an environment } e_2 \in k(c, \(\tilde{i}\), \(\tilde{o}\), e_1, r_1) \cap \varphi. \text{ By definition of } k, \text{ it then follows that there exists a result } r_2 \text{ such that } \\
\langle c, e_2 \rangle \Downarrow r_2 \wedge e_1 \overset{i}{\sim} e_2 \wedge r_1 \overset{o}{\sim} r_2. \text{ Since } e_2 \in \varphi, e_2 \not\in \varphi \text{ also holds, satisfying opacity.}
\]}

\[\text{Instantiations}\]

\[\text{Proof of Theorem 1. Clearly } \sim_\ell \text{ and } \approx_\ell \text{ are reflexive and } =_\ell \text{ is an equivalence. Therefore, the statement follows from Lemma 4.} \]

\[\text{Proof of Theorem 2. Clearly } \sim_\ell, \approx_\ell, \text{ and } \cong_\ell \text{ are reflexive and } =_\ell \text{ is an equivalence. Therefore, the statement follows from Lemma 4.} \]

\[\text{Proof of Theorem 3. It easily follows that } =_E^\ell \text{ is an equivalence. Since } DR(\(\tilde{\sim}\), \ell, E) = NI(\(=E^\ell\), \(\tilde{\sim}\)) \text{ the statement then follows from Lemma 4.} \]
Chapter 1: Opacity

Enforcement

We denote the first and second projections of a pair \( p \) by and \( \pi_1(p) \) and \( \pi_2(p) \) respectively. Two memories \( m_1, m_2 \) are equal on a set of expressions \( E \) (denoted by \( m_1 =_E m_2 \)) if \( \forall e \in E. m_1(e) = m_2(e) \).

For a set of expressions \( E \) we define the set \( C(E) \) where constant (integer) expressions are removed by \( C(E) = E \setminus \mathbb{Z} \).

**Lemma 8.** For any set of expressions \( E \), \( m_1 =_E m_2 \) holds if and only if \( m_1 =_{C(E)} m_2 \).

**Proof.** We prove this by contradiction: \( \Rightarrow \): Assume \( m_1 \neq_{C(E)} m_2 \) and \( m_1 =_E m_2 \). Hence there exists an expression \( e \in C(E) \setminus E \) such that \( m_1(e) \neq m_2(e) \). However, this is a contradiction since clearly \( C(E) \subseteq E \).

\( \Leftarrow \): Assume \( m_1 \neq_E m_2 \) and \( m_1 =_{C(E)} m_2 \). Hence there exists an expression \( e \in E \setminus C(E) \) such that \( m_1(e) \neq m_2(e) \). Since \( e \in E \setminus C(E) \), it must also hold that \( e \in \mathbb{Z} \). Then however, by the semantics of expression evaluation, we get \( m_1(e) = m_2(e) \), contradicting \( m_1(e) \neq m_2(e) \).

We can now prove Lemma 7 (restated here for convenience):

**Lemma 7.** Whenever \( \langle c, m_1, M, \delta_0 \rangle \xrightarrow{t_1}^* \langle c', m'_1, M', \delta' \rangle \) and \( m_2 \in M' \), then there exist \( t_2, m'_2 \) such that:

\[
\langle c, m_2 \rangle \xrightarrow{t_2}^* \langle c', m'_2 \rangle \land t_1 \cong_L t_2 \tag{I}
\]

and

\[
\forall e \in E. m_1(\pi_1(\delta(e, m'_1))) = m'_1(e) \tag{II}
\]

and

\[
\forall e \in E. m_1(\pi_2(\delta(e, m'_1))) = m'_1(e) \Rightarrow m_2(\pi_1(\delta(e, m'_1))) = m'_2(e) \tag{III}
\]

where \( \delta_0(x) = (x, \emptyset) \) and \( \delta_0(a[i]) = (a[i], \emptyset) \).

**Proof.** We prove the lemma by induction over \( \langle c, m_1, M, \delta_0 \rangle \xrightarrow{t_1}^* \langle c', m'_1, m', \delta' \rangle \):

Reflexive case: In this case we have \( c' = c, m'_1 = m_1, t_1 = \langle \rangle, M' = M, \delta' = \delta_0 \). We set \( m'_2 = m_2 \) and \( t_2 = \langle \rangle \). Clearly, (I) is satisfied.
For (II), let \( e \in \mathbb{E} \). The conclusion follows easily since 
\( \pi_1(\delta_0(e, m_1)) = e \) and hence \( m_1(\pi_1(\delta_0(e, m_1))) = m_1(e) \). 
(III) follows analogously.

Transitive case: Assume \( \langle c, m_1, M, \delta \rangle \xrightarrow{t_1} \langle c', m', M', \delta' \rangle \) and 
\( \langle c', m'_1, M', \delta' \rangle \xrightarrow{t'_1} \langle c'', m''_1, M'', \delta'' \rangle \).

First note that \( M'' \subseteq M' \) holds for all evaluation rules. We 
then apply the induction hypothesis to obtain \( t_2, m'_2 \) such that:

\[
\langle c, m_2 \rangle \xrightarrow{t_2} \langle c', m'_2 \rangle \wedge t_1 \approx_L t_2 \quad \text{(IH1)}
\]

and

\[
\forall e \in \mathbb{E}. m_1(\pi_1(\delta'(e, m'_1))) = m'_1(e) \quad \text{(IH2)}
\]

and

\[
\forall e. m_1 =_{\pi_2(\delta'(e, m'_1))} m_2 \Rightarrow m_2(\pi_1(\delta'(e, m'_2))) = m'_2(e) \quad \text{(IH3)}
\]

We proceed by induction on \( \langle c', m'_1, M', \delta' \rangle \xrightarrow{t'_1} \langle c'', m''_1, M'', \delta'' \rangle \):

Case E-Skip: By the semantics of (instrumented) evaluation 
we get that \( M'' = M', c'' = e, m''_1 = m'_1, \delta'' = \delta' \). Hence, (II), and 
(III) follow directly from IH1, IH2, and IH3. (I) follows easily by

setting \( m''_2 = m'_2 \) and \( t'_2 = t'_2 \).

Case E-Assign: \( c' = x := e' \). The evaluation semantics yield 
\( m''_1 = m'_1[x \mapsto m'_1(e')] \), \( c'' = e, \delta'' = \delta'[x \mapsto \delta'(e', m'_1)] \) and \( M'' = M' \).

For (I), set \( t'_2 = t_2 \) and \( m''_2 = m'_2[x \mapsto m'_2(e')] \).

For (II) and (III), let \( e \in \mathbb{E} \). We proceed by induction over \( e \): 
The cases for integer and boolean literals follow trivially.

Case \( e = y, y \neq x \): First we note that 
\( \delta''(y, m''_1) = \delta'(y) = \delta'(y, m'_1) \)

We then conclude using IH2 and IH3.

Case \( e = x \): We note that 
\( \delta''(x, m''_1) = \delta'(x) = \delta'(x, m'_1) \)

For (II), we then conclude as follows:

\[
m_1(\pi_1(\delta'(x, m'_1))) = m_1(\delta'(e', m'_1)) \xrightarrow{\text{IH2}} m'_1(e') = m''_1(x)
\]
For (III), assume \( m_1 = \pi_2(\delta''(x, m''_1)) \) \( m_2 \). Hence \( m_1 = \pi_2(\delta'(e, m'_1)) \) \( m_2 \). Using IH3 we then conclude by:

\[
m_2(\pi_1(\delta''(x, m''_1))) = m_2(\pi_1(\delta'(e, m'_1))) \\
\text{IH} = m_2(e') = m''_2(x)
\]

Case \( e = e_1 \oplus e_2 \) (where \( \oplus \) is one of the binary operators in the expression language): By induction hypothesis, it holds that \( m_1(\pi_1(\delta''(e_k, m''_1))) = m''_1(e_k) \) and \( m_1 = \pi_2(\delta''(e_k, m''_1)) m_2 \Rightarrow \)

\[
m_2(\pi_1(\delta''(e_k, m''_1))) = m_2(e_k) \quad \text{(for } k = 1, 2). \]

For (II), we conclude by the semantics of expression evaluation.

For (III), assume \( m_1 = \pi_2(\delta''(e, m'_1)) m_2 \). By definition of \( \delta''(\cdot, \cdot) \), this implies

\[
m_1 = \pi_2(\delta''(e_1, m''_1)) \cup \pi_2(\delta''(e_2, m''_2)) m_2 \quad \text{and hence also } m_1 = \pi_2(\delta''(e_k, m''_1)) m_2 \quad \text{for } k = 1, 2. \]

We then obtain the desired conclusion using IH3 and the semantics of expression evaluation.

The cases for pair expressions and projections and if expressions follow analogously.

Case \( e = a[e_i] \): The induction hypothesis yields

\[
m_1(\pi_1(\delta''(e_i, m''_1))) = m''_1(e_i) \quad \text{and } m_1 = \pi_2(\delta''(e_i, m''_1)) m_2 \Rightarrow \]

\[
m_2(\pi_1(\delta''(e_i, m''_1))) = m''_2(e_i). \]

We also note, using the definition of \( \delta''(\cdot, \cdot) \) and \( \delta'' \), that

\[
\delta''(a[e_i], m'_1) \\
= (\pi_1(\delta''(a[m''_1(e_i)])), \delta''(e_i, m''_1)) \cup \pi_2(\delta''(a[m''_1(e_i)])) \\
= (\pi_1(\delta'(a[m''_1(e_i)])), \delta''(e_i, m''_1)) \cup \pi_2(\delta'(a[m''_1(e_i)]))
\]

We write \( \delta'(e, m) \cup E \) as an abbreviation for \( \{\pi_1(\delta'(e, m))\} \cup \pi_2(\delta'(e, m) \cup E) \).

Hence it holds that \( \pi_1(\delta''(a[e_i], m''_1)) = \pi_1(\delta'(a[m''_1(e_i)], m'_1)); \)

note that \( a[m''_1(e_i)] \) is an array-lookup expression with a constant as the index expression, which allows changing the second parameter of \( \delta' \) to \( m'_1 \). Hence, for (II), we compute:

\[
m_1(\pi_1(\delta''(a[e_i], m''_1))) = m_1(\pi_1(\delta'(a[m''_1(e_i)], m'_1))) \\
\text{IH} = m'_1(a[m''_1(e_i)]) \\
= m''_1(a[m''_1(e_i)]) \\
= m''_1(a[e_i])
\]
For \((III)\), assume \(m_1 = \pi_2(\delta''(a[e_i], m''_1)) m_2\). Note from the previous result that \(C(\pi_2(\delta'(a[m''_1(e_i)], m'_1))) \subseteq \pi_2(\delta''(a[e_i], m''_1))\).

Using Lemma 8, this yields \(m_1 = \pi_2(\delta'(a[m''_1(e_i)], m'_1)) m_2\). Combined with IH3 we then obtain

\[
m_2(\pi_1(\delta'(a[m''_1(e_i)], m'_1))) = m'_2(\pi_1(m''_1(e_i)))
\]

(1.1)

Moreover, since \(\pi_2(\delta''(e_i, m''_1)) \subseteq \pi_2(\delta''(a[e_i], m''_1))\), we also note, using the induction hypothesis for \(e_i\), that \(m_2(\pi_1(\delta''(e_i, m''_1))) = m''_2(e_i)\). Together with \(m_1(\pi_1(\delta''(e_i, m''_1))) = m'_1(e_i)\) and the previous result for \(\delta''(a[e_i], m''_1)\), we then obtain

\[
m''_1(e_i) = m''_2(e_i)
\]

(1.2)

Hence we conclude:

\[
m_2(\pi_1(\delta''(a[e_i], m''_1))) = m_2(\pi_1(\delta'(a[m''_1(e_i)], m'_1)))
\]

\[
\overset{1.1}{=} m'_2(a[m''_1(e_i)])
\]

\[
\overset{1.2}{=} m'_2(a[m''_2(e_i)])
\]

\[
= m''_2(a[m''_2(e_i)])
\]

\[
= m''_2(a[e_i])
\]

Case E-ARRASSIGN: By the evaluation semantics it holds that 
\(c' = a[e_i] := e', m''_1 = m'_1[a \mapsto m'_1(a)[m'_1(e_i) \mapsto m'_1(e')]], M'' = M' \cup [m_1|\delta'(e_i, m'_1)], \delta'' = \delta'[a[m'_1(e_i)] \mapsto \delta'(e', m'_1)], \) and \(c'' = e\).

(I) follows easily by setting \(m''_2 = m'_2[a \mapsto m'_2(a)[m'_2(e_i) \mapsto m'_2(e')]]\) and \(t''_2 = t_2\).

We for \((II)\) and \((III)\) let \(e \in E\). We proceed by induction over \(e\):

The cases for integer and boolean literals follow again easily.

Case \(e = x\): In this case it holds that \(\delta''(x, m''_1) = \delta'(x) = \delta'(x, m'_1)\) and hence the conclusion follows from IH2 and IH3.

Cases \(e_1 \oplus e_2\), pairs, projections, and if expressions follow as for case E-ASSIGN.

Case \(e = a[e_j]\): By induction hypothesis for \(e_j\), we obtain

\[
m_1(\pi_1(\delta''(e_j, m''_1))) = m''_1(e_j)
\]

(IH' 1)

and

\[
m_1 = \pi_2(\delta''(e_j, m''_1)) m_2 \Rightarrow m_2(\pi_1(\delta''(e_j, m''_1))) = m''_2(e_j)
\]

(IH' 2)
Since $m_2 \in M''$, we obtain that:

$$m_1 = \delta'(e_i, m'_1) m_2$$  \hspace{1cm} (1.3)

Using IH$_3$, this yields:

$$m_2(\pi_1(\delta'(e_i, m'_1))) = m'_2(e_i)$$  \hspace{1cm} (1.4)

Moreover, 1.3 also implies that $m_2(\pi_1(\delta'(e_i, m'_1))) = m_1(\pi_1(\delta'(e_i, m'_1)))$. Hence:

$$m'_2(e_i) \overset{1.4}{=} m_2(\pi_1(\delta'(e_i, m'_1)))$$

$$= m_1(\pi_1(\delta'(e_i, m'_1))) \overset{\text{IH}_2}{=} m'_1(e_i)$$  \hspace{1cm} (1.5)

We distinguish the cases $m''_1(e_j) = m'_1(e_i)$ and $m''_1(e_j) \neq m'_1(e_i)$.

Assume first that $m''_1(e_j) = m'_1(e_i)$.

We first note:

$$(\delta''(a[e_j], m''_1)) =$$
$$(\pi_1(\delta''(a[m''_1(e_j)])), \delta''(e_j, m''_1) \cup \pi_2(\delta''(a[m''_1(e_j)]))) =$$
$$(\pi_1(\delta''(a[m'_1(e_i)])), \delta''(e_j, m''_1) \cup \pi_2(\delta''(a[m'_1(e_i)]))) =$$
$$(\pi_1(\delta'(e', m'_1)), \delta''(e_j, m''_1) \cup \pi_2(\delta'(e', m'_1)))$$

Therefore:

$$\pi_1(\delta''(a[e_j], m''_1)) = \pi_1(\delta'(e', m'_1)) \land$$  \hspace{1cm} (1.6)

$$\pi_2(\delta'(e', m'_1)) \subseteq \pi_2(\delta'(a[e_j], m''_1))$$  \hspace{1cm} (1.7)

For $(II)$, we compute:

$$m_1(\pi_1(\delta''(a[e_j], m''_1))) \overset{1.6}{=} m_1(\pi_1(\delta'(e', m'_1)))$$

$$\overset{\text{IH}_2}{=} m'_1(e')$$

$$= m''_1(a)(m'_1(e_i))$$

$$= m''_1(a)(m''_1(e_j))$$

$$= m''_1(a[e_j])$$

For $(III)$, assume also that

$$m_1 = \pi_2(\delta''(a[e_j], m''_1)) m_2$$  \hspace{1cm} (1.8)
With IH$_3$ and 1.7, this yields
\[ m_2(\pi_1(\delta'(e', m'_1))) = m'_2(e') \tag{1.9} \]

Using 1.8 we also obtain \( m_1 = \pi_2(\delta''(e_j, m''_1)) \) \( m_2 \) and with IH'_2 this yields:
\[ m_2(\pi_1(\delta''(e_j, m''_1))) = m''_2(e_j) \tag{1.10} \]

We then also compute:
\[ m''_1(e_j) \overset{IH'_1}{=} m_1(\pi_1(\delta''(e_j, m''_1))) \overset{1.8}{=} m_2(\pi_1(\delta''(e_j, m''_1))) \overset{1.10}{=} m''_2(e_j) \tag{1.11} \]

With \( m''_1(e_j) = m'_1(e_i) \) it then follows that \( m''_2(e_j) = m'_1(e'_i) \)
and with 1.5 we obtain \( m''_2(e_j) = m'_2(e_i) \).

We then conclude as follows:
\[ m_2(\pi_1(\delta''(a[e_j], m''_1))) \overset{1.6}{=} m_2(\pi_1(\delta'(e', m'_1))) \overset{1.9}{=} m'_2(e') = m''_2(a)(m'_2(e_i)) = m''_2(a)(m''_2(e_j)) = m''_2(a[e_j]) \]

Now assume that
\[ m''_1(e_j) \neq m'_1(e_i) \tag{1.12} \]

Note that
\[ \delta''(a[e_j], m''_1) = \]
\[ (\pi_1(\delta''(a[m''_1(e_j)])), \delta''(e_j, m''_1) \cup \pi_2(\delta''(a[m''_1(e_j)])) = \]
\[ (\pi_1(\delta'(a[m'_1(e_j)])), \delta''(e_j, m''_1) \cup \pi_2(\delta'(a[m'_1(e_j)])) \]

Hence:
\[ \pi_1(\delta''(a[e_j], m''_1)) = \pi_1(\delta'(a[m''_1(e_j)], m'_1)) \tag{1.13} \]

and
\[ C(\pi_2(\delta'(a[m''_1(e_j)], m'_1))) \subseteq \pi_2(\delta''(a[e_j], m''_1)) \tag{1.14} \]
For (II), we compute:

\[ m_1(\pi_1(\delta''(a[e_j], m_1''(e_j))))^{.13} = m_1(\pi_1(\delta'(a[m_1''(e_j)], m_1')))) \]

\[ IH'_1 \]

\[ = m_1'(a[m_1''(e_j)]) \]

\[ ^{.12} = m_1''(a[m_1''(e_j)]) \]

\[ = m_1''(a[e_j]) \]

For (III), assume again that

\[ m_1 = \pi_2(\delta''(a[e_j], m_1'')) \]

From 1.15 we also obtain

\[ m_1 = \pi_2(\delta''(a[e_j], m_1'')) \]

and with IH'2, this yields

\[ m_2(\pi_1(\delta''(e_j, m_1''))) = m_2''(e_j). \]

1.15 also yields

\[ m_1 = \pi_1(\delta''(e_j, m_1'')) \]

and hence we conclude, with IH'_1, that

\[ m_1''(e_j) = m_2''(e_j) \]

Therefore, we also obtain

\[ m_2''(e_j) \neq m_2'(e_i) \]

Using 1.14, IH_3, and 8 we get:

\[ m_2(\pi_1(\delta'(a[m_1''(e_j)], m_1'))) = m_2'(a[m_2''(e_j)]) \]

Hence we conclude by:

\[ m_2(\pi_1(\delta''(a[e_j], m_1'')))^{.13} = m_2(\pi_1(\delta'(a[m_1''(e_j)], m_1'))) \]

\[ IH'_2 \]

\[ = m_2'(a[m_1''(e_j)]) \]

\[ ^{.16} = m_2'(a[m_2''(e_j)]) \]

\[ ^{.17} = m_2'(a[m_2''(e_j)]) \]

\[ = m_2''(a[e_j]) \]

Case \( e = b[e_j] \): Analogous to case \( m_1''(e_j) \neq m_1'(e_i) \) for \( e = a[e_j] \).

Case E-OUT: \( c' = \textbf{out} \) \( e \), \( m_1'' = m_1', \delta'' = \delta', c'' = \varepsilon. \) We distinguish the cases \( \ell = L \) and \( \ell = H \).

If \( \ell = H \), set \( m_2'' = m_2' \) and \( t_2' = t_2 \) to show (I), (II) and (III) follow from the induction hypothesis.
If ℓ = L, set \( m''_2 = m_2' \) and \( t_2 = t_2.m'_1(e) \). We now show that \( m'_1(e) = m'_2(e) \): Since \( M'' = M' \cap [m_1]\delta'(e, m'_1) \), we have that \( m_1 = \tau_2(\delta'(e, m'_1)) \) \( m_2 \). By IH2 and IH3, we obtain \( m_1(\pi_1(\delta'(e, m'_1))) = m'_1(e) \) and \( m_2(\pi_1(\delta'(e, m'_1))) = m'_2(e) \). Hence we conclude \( m'_1(e) = m'_2(e) \), satisfying (I).

(II) and (III) again follow from the induction hypothesis.

Cases E-IfTrue, E-IFFalse: \( c' = \text{if } e \{ c_1 \} \text{ else } \{ c_2 \} \), \( m''_1 = m'_1, \delta'' = \delta', M'' = M \cap [m_1]\delta'(e, m'_1) \). Moreover, there exists an \( i \in \{1, 2\} \) such that \( c'' = c_i \). Since \( m_2 \in M'' \), it holds that \( m_1 = \delta'(e, m'_1) \) \( m_2 \). From IH2 and IH3 we then conclude that \( m'_1(e) = m'_2(e) \). Hence \( \langle c', m'_2 \rangle \xrightarrow{\delta} \langle c_i, m'_2 \rangle \), satisfying (I).

(II) and (III) follow from IH2 and IH3 since \( \delta'' = \delta' \).

Case E-WhileTrue: \( c' = \text{while } e \text{ do } c_1, m''_1 = m'_1, \delta'' = \delta', M'' = M' \cap [m_1]\delta'(e, m'_1), c'' = c_1; \text{ while } e \text{ do } c_1 \). Analogously to the previous cases, \( m_2 \in M'' \) implies \( m'_1(e) = m'_2(e) \). Hence \( \langle \text{while } e \text{ do } c_1, m'_2 \rangle \xrightarrow{\delta} \langle c_1; \text{ while } e \text{ do } c_1, m'_2 \rangle \), satisfying (I).

Case E-WhileFalse: Like E-WhileTrue.

Cases E-Seq, E-SeqEmpty: Immediate from induction hypothesis for sub-commands.

\[ \square \]

**Lemma 9.** If \( \tau(m_1, c, M, \delta) = (M', \delta') \), then \( M' \subseteq M \).

**Proof.** Immediate from definition of \( \tau \). \( \square \)

**Proof of Theorem 4.** Direct consequence from Lemmas 7 and 9. \( \square \)

**Lemma 10.** If \( \langle c, m \rangle \xrightarrow{t_1}^* \langle c_1, m_1 \rangle \) and \( \langle c, m \rangle \xrightarrow{t_2}^* \langle c_2, m_2 \rangle \), then \( \exists t'_1. \langle c_1, m_1 \rangle \xrightarrow{t'_1}^* \langle c_2, m_2 \rangle \land t_2 = t_1.t'_1 \) or \( \exists t'_2. \langle c_2, m_2 \rangle \xrightarrow{t'_2}^* \langle c_1, m_1 \rangle \land t_1 = t_2.t'_2 \).

**Proof.** We prove this by contradiction.

Assume \( \langle c_0, m_0 \rangle \xrightarrow{t_1} \langle c_1, m_1 \rangle \ldots \xrightarrow{t_n} \langle c_n, m_n \rangle \) and \( \langle c_0, m_0 \rangle \xrightarrow{t'_1} \langle c'_1, m'_1 \rangle \ldots \xrightarrow{t'_n} \langle c'_1, m'_1 \rangle \). Assume furthermore that neither

\[ \exists t. \langle c_n, m_n \rangle \xrightarrow{t} \langle c'_i, m'_i \rangle \land t'_1 \ldots t'_i = t_1 \ldots t_n.t \]

nor

\[ \exists t. \langle c'_i, m'_i \rangle \xrightarrow{t'} \langle c_n, m_n \rangle \land t_1 \ldots t_n = t'_1 \ldots t'_i.t' \]

holds.

Without loss of generality, let \( l \leq n \).
We first show that there exists an \(1 \leq i \leq l\) such that 
\[c_i \neq c_i' \lor m_i \neq m_i' \lor t_i \neq t_i'.\]
If no such \(i\) exists, then \(c_k = c_k' \land m_k = m_k' \land t_k = t_k'\) for all \(k \leq l\). Then however, we have

\[
\langle c_i, m_i \rangle = \langle c_i, m_i \rangle \xrightarrow{t_{i+1}, \ldots, t_n}^* \langle c_n, m_n \rangle
\]

contradicting our assumption. Hence such an \(i\) must exist.

Let \(i\) be the smallest natural number that satisfies this. This implies
\[
\langle c_{i-1}, m_{i-1} \rangle \xrightarrow{t_i} \langle c_i, m_i \rangle \text{ and } \langle c_{i-1}, m_{i-1} \rangle \xrightarrow{t_i'} \langle c_i', m_i' \rangle
\]
where \(c_i \neq c_i' \lor m_i \neq m_i' \lor t_i \neq t_i'.\)

However, since single evaluation steps are deterministic, we have \(c_i = c_i' \land m_i = m_i' \land t_i = t_i'\), contradicting the property of \(i\).

\[\square\]

**Proof of Theorem 5.** Assume 
\[
\langle c, m_1, [m_1]_\varphi \cap \varphi, id \rangle \xrightarrow{t_1}^* \langle c_1', m_1', M', \delta' \rangle, \langle c, m_2 \rangle \xrightarrow{t_2}^* \langle c_2', m_2' \rangle, \text{ and } m_2 \in M'.
\]

By Lemma 7, there exist \(t_2', m_2''\) such that \(\langle c, m_2 \rangle \xrightarrow{t_2'} \langle c_1', m_1'' \rangle \land t_1 \approx \ell t_2'.\) From Lemma 9, we also get \(m_2 \notin \varphi\).

Using Lemma 10, it must hold that \(\langle c_1', m_2'' \rangle \xrightarrow{t_2''} \langle c_2', m_2' \rangle\) for some \(t_2''\) such that \(t_2 = t_2', t_2''\) (in the other case of Lemma 10, we conclude analogously). Since \(t_1 \approx \ell t_2'\), the traces \(t_1\) and \(t_2\) agree on low events for prefixes \(t_1\) and \(t_2'\). Hence, \(t_1 \sim \ell t_2\) holds, as desired.

\[\square\]

**Proof of Theorem 6.** If the algorithm returns a pair \((m_2, t_2)\) where \(t_2\) is the trace produced by \(\langle c, m_2 \rangle\), then, by assumption on the sampling function \(S\), it holds that \(m_2 \in [m_1]_{=\ell}\).

Moreover, by construction of the algorithm, we have \(t_1 \sim \ell t_2\) and \(m_2 \notin \varphi\), as these conditions are ensured before returning \((m_2, t_2)\).

\[\square\]
Explicit Secrecy: A Policy for Taint Tracking

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

Abstract. 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 popular dynamic and static taint trackers.
2.1 Introduction

_Taint tracking_ is a striking success story in computer security. It is used to enhance both confidentiality and integrity in a wide variety of applications ranging from hardware-level [22, 55] and binary-level tainting [21, 53] to tainting in mobile [33] and web [45, 58, 42] applications. Languages such as Perl [4] and Ruby [3] have built-in support for taint tracking, and extensions for languages such as Java [36, 58], JavaScript [45], and Python [27] are available to perform taint tracking.

**Motivation.** At its heart, taint tracking is about tracking data dependencies as data is propagated by the system. In the setting of a mobile app, taint tracking can help detect privacy leaks, e.g., when the app attempts to send the user’s location to a third-party. This is done by labeling location data as secret and detecting when secret-labeled data is sent to third parties over the network, as e.g. in the popular TaintDroid tool [33]. This is an example of enforcing a confidentiality policy by taint tracking. Or, in the setting of a web application, some input sources (e.g., user input) can be labeled as “tainted,” and taint tracking can be used to prevent tainted data from affecting sensitive sinks (e.g., writing to a system file or generating HTML for a web page) [58, 42]. This is an example of enforcing an integrity policy by taint tracking.

A key reason behind the success of taint tracking tools in practice is that taint tracking is a pure data dependency analysis. It only tracks _explicit_ [31] flows of the form \( l := h + 1 \) where the value of \( h \) is explicitly leaked into \( l \), while ignoring _implicit_ [31] flows of the form \( \text{if } h \text{ then } l := 1 \text{ else } l := 0 \) via the control-flow structure of the program. Ignoring implicit flows makes taint trackers unsound (especially for malicious code where the attacker is in control of what flows to exploit [49]), but this loss of soundness is compensated by a large increase in practicality because the enforcement need not track control flows. This makes a crucial difference for the practicality because tracking control flow is hard: although much progress has been made on information-flow tracking, dealing with control flow in expressive programming languages remains challenging [50], especially in dynamic languages as JavaScript where it is hard to predict side effects along alternative execution paths [37]. By comparison, taint tracking is appealingly lightweight and effec-
tive, as evidenced by its success in finding vulnerabilities in real systems and its adoption in many programming languages.

The success of taint tracking brings us to a seemingly basic yet highly elusive question: What, precisely, is taint tracking good for? In other words, what is the formal meaning of “data dependency” and “explicit flows,” and what is an appropriate soundness criterion for a taint tracker? Answers to these questions are not to be found in program analysis literature [47], where the focus is not on data dependencies themselves but on their use for code transformation and optimization [6]. What is more surprising is that general answers to these basic questions, to the best of our knowledge, are neither to be found in the security literature!

This is in sharp contrast to the general area of information flow [50], which is concerned with specifying confidentiality and integrity for both explicit and implicit flows, where there is a large body of work on policies ranging from noninterference [26, 34], which allows no dependencies from secret sources to public sinks, to various flavors of declassification [51] and endorsement [10] policies.

It is thus highly desirable to improve the understanding of explicit (vs. implicit) flows and develop a general policy framework for taint tracking that can be related to known security characterizations and serve as a target condition for taint tracking mechanisms.

**Background.** To date, efforts on characterizing tainting policies have been scarce. Rather than specifying a security condition, it is common to state desired properties of the enforcement, phrased in an enforcement-centric fashion, as, e.g., in graph-based properties in the work by Livshits and Chong [41, 42]. Another approach is to formalize the essence of taint tracking by formalizing a generic taint tracker, as, e.g., in the work by Schwartz et al. [53]. While succintly representing what happens inside such tools as BitBlaze [55] and BAP [18], such a formalization is inherently low-level [53].

What we are looking for is an enforcement-independent condition that captures the essence of explicit flows and that can be checked against independently interesting approaches to enforcement.

Closest to our needs is Volpano’s weak secrecy [62], the only policy we are aware of that focuses on describing what can be enforced by explicit flow analysis. Weak secrecy makes use of the classical information-flow notion of noninterference [26, 34], which
demands that a program’s secret input may not influence the program’s public output. Intuitively, a program satisfies weak secrecy if, for any run of a program up to a given point in time, the sequence of assignments performed by the program satisfies noninterference.

While weak secrecy is a reasonable starting point, there are some roadblocks for adopting it for reasoning about practical taint trackers. The fact that the definition is syntactic in nature, relying on extracting assignment commands from the original program, implies that the definition does not scale to languages with rich features such as reflection. This makes adapting it to low-level languages particularly challenging. Low-level machines allow “reflective” programming idioms such as programs that read or write their own instructions. A direct extension of weak secrecy in this setting does not work because a modified program will not necessarily have the same instructions in the memory. Also, low-level machines typically have many instructions, and each instruction may have complex semantics (e.g., jump-and-link instructions that modify both control flow and memory). There is no clear criterion for how to extend the weak secrecy definition. This is particularly concerning, given that taint tracking often targets low-level machines [21, 53]. Other features that challenge weak secrecy, even in high-level languages, include expressions with side effects, which require custom-tailored encodings to get weak secrecy to work. Finally, the definition is indirect, defining a weak policy, weak secrecy, via the stronger policy of noninterference.

What we want is a language-independent, semantic definition of “correct taint tracking”, generalizing weak secrecy and applicable to a wide range of models from high-level languages to low-level machines. Our goal is to lay foundation for exploring the design space of ways to split program configurations into data and control.

Contributions. Motivated by the above, we propose a general semantic framework for specifying explicit flows, offering the following contributions: (i) We propose a knowledge-based semantic security condition, explicit secrecy, that captures the essence of explicit flows in a language-independent way, based on a distinction between “control” and “data” that is specified by the language designer. Intuitively, explicit secrecy separates data and control and demands the security of flows for the data part. (ii) We show the flexibility of the model by incorporating possibilities of declas-
sification and sanitization, unexplored features in the context of previous attempts to give a soundness condition to taint tracking.

(iii) We instantiate explicit secrecy for a high-level imperative language with I/O to obtain a soundness criterion for taint tracking.

(iv) We instantiate explicit secrecy for a simple RISC machine to yield a soundness criterion in the setting of low-level languages.

(v) Being a semantic condition, explicit security can readily be related to known security characterizations. We show that its instantiation to a simple imperative language agrees with weak secrecy, demonstrating that explicit secrecy indeed generalizes weak secrecy. Thus, like weak secrecy, explicit secrecy does not subsume (and is not subsumed by) noninterference. Further, we establish that explicit secrecy is stronger than an intuitive declassification-based condition (dubbed Control-Flow Gradual Release) that declassifies the guards at branching points.

(vi) We use explicit secrecy to illuminate the behavior of real-world taint trackers by analyzing the core of taint tracking engines from several popular languages and tools. In particular: we show soundness for a dynamic enforcement mechanism for a simple subset of high-level languages such as Perl and Ruby; we show soundness for a dynamic enforcement mechanism [53] underlying the BAP [18] and BitBlaze [55] tools for low-level machines; and we define a simple static enforcement mechanism reminiscent of the ones in Andromeda [59] and Flow-Droid [7] and prove it sound.

Scope. The paper lays groundwork for judging the soundness of existing and new enforcement mechanisms. We evaluate the foundational framework on the core mechanisms underlying practical enforcement techniques. The approach gives benefit whether the soundness proofs succeed (by giving assurance of what is achieved) or fail (by pointing to insecurities). Thus, in addition to the foundational impact, the practical impact of the paper is the demonstration that core mechanisms of the practical tools (analyzed in Section 2.3) are sound. While experimental evaluation of the approach is important, it can only be done once a sound foundation is in place. Similarly to other foundational work on security (e.g., [43, 11, 24, 53]), we note that while the experimental evaluation is not in the scope of the present study, it is subject to subsequent engineering work that can build on the foundations.

For simplicity, we focus on confidentiality in the rest of the pa-
per. Note that noninterference is suitable for both integrity (e.g., tracking buffer overruns) and confidentiality (e.g., tracking leaks in mobile apps). These are dual properties in terms of information flow [16]. Similarly, our notion of explicit secrecy is equally suitable for both integrity and confidentiality, through dualization. Using the duality, we can interpret tainted sources/sinks as untrusted and reason about attacker influence similarly to knowledge-based approaches. Sanitization functions for preventing injection vulnerabilities can be modeled as declassification/endorsement.

2.2 Specifying Explicit Flows

We first review the definition of weak secrecy and elaborate on difficulties with instantiating weak secrecy to richer languages (Section 2.2.1). To address these difficulties, we introduce explicit secrecy (Section 2.2.2), a language-agnostic security property that formalizes the idea of “security with respect to explicit flows only.” We show how to instantiate this property to both a simple imperative language (Section 2.2.4) and machine code for a simple RISC machine (Section 2.2.4). To benchmark explicit secrecy against information-flow conditions, we show examples demonstrating that it is incomparable in power to noninterference. Further, we explore an intuitive approach to characterize explicit-flow tracking using a variant of gradual release [11], a knowledge-based information release policy. We show that explicit secrecy is stronger than this characterization (Section 2.2.5).

2.2.1 Weak Secrecy

To formalize security with respect to (just) explicit flows, Volpano [62] introduced weak secrecy for a simple imperative language. We begin by recapitulating his definition.

Our language includes global variables, while loops, conditionals, assignment, and output.

\[
c ::= \text{skip} \mid c_1; c_2 \mid x := e \mid \text{out } e \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
\]

We assume a set of variables \( \text{Var} \). A configuration \((c, m)\) consists of a command \(c\) and a memory \(m \in \text{Mem}\) mapping variables to integers, i.e. \(\text{Mem} = \text{Var} \to \mathbb{Z}\). We write \((c, m) \xrightarrow{\alpha} (c', m')\) to denote
that a configuration \((c, m)\) evaluates in one step to configuration \((c', m')\) while producing trace \(\alpha \in \text{Obs}^*\), where \(\text{Obs} = \mathbb{Z}\). The (standard) definition of this relation appears in Appendix 2.A.1. A terminated program is represented by \(\varepsilon\). We also write \(\varepsilon\) for the empty trace.

Clark and Hunt [23] show that for the security of deterministic programs, such as programs in the presented language, it makes no difference whether the environments are modeled as streams or strategies. A further common simplification [9] is to model input to a program by initial memories.

We assume that each variable \(x\) has an associated security level \(\Gamma(x) \in \mathcal{L}\), where \((\mathcal{L}, \sqcap, \sqsubseteq)\) is a lattice of security levels. In the examples, we assume a two-level security lattice consisting of \(H\) for variables containing confidential information and \(L\) for variables containing public information, with \(L \sqsubseteq H\). Two memories \(m_1\) and \(m_2\) are said to be low equivalent, written \(m_1 =_L m_2\), iff \(\forall x. \Gamma(x) = L \Rightarrow m_1(x) = m_2(x)\).

Our language differs in one small respect from the one used by Volpano: there, every assignment to a low variable generates an event that is visible to the attacker. While this allows for content in high variables to be updated with low values, any assignment of high content to low variables renders the program insecure. We make the setup a bit more flexible by limiting the attacker to observing the program’s external behavior, introducing explicit output instructions that generate attacker-visible events, with assignments not being directly observable. However, this choice is orthogonal to the security conditions presented here and in Section 2.2.2.

The intuition of weak secrecy is that every possible sequence of non-control-flow statements (assignments and outputs) executed by a program run has to be noninterfering. To define this, we annotate the evaluation rules in the semantics to record executed explicit flow statements, writing \((c, m) \xrightarrow[\alpha]{d} (c', m')\) if the step from \((c, m)\) to \((c', m')\) generates the explicit flow statement \(d\) (and observable events \(\alpha\)). Note that \(d\) is distinct from \(\alpha\)—i.e., it is not part of the trace of attacker-visible observations generated by the program. For example, here are the instrumented rules for assignment, output, and the true case of the conditional\(^1\) (the appendix gives the other

\(^1\)We assume that an expression such as \(e = 0\) evaluates to 1 if \(m(e) = 0\) and
rules); when $d$ and/or $\alpha$ are empty or unimportant, we omit the superscript and/or subscript on the $\rightarrow$:

\[
m(e) = n \quad (x := e, m) \xrightarrow{x := e} (\varepsilon, m[x \mapsto n]) \quad (\text{E-Assign})
\]

\[
m(e) = n \quad (\text{out } e, m) \xrightarrow{n \text{ out } e} (\varepsilon, m) \quad (\text{E-Out})
\]

\[
m(e = 0) = n \quad n = 0 \quad (\text{if } e \text{ then } c_1 \text{ else } c_2, m) \rightarrow (c_1, m) \quad (\text{E-IfTrue})
\]

Note that the notion of “control-flow statement” is not a formal one and is left open when instantiating weak secrecy to another language. While for this language it is reasonably straight-forward to determine which statements affect the memory, more advanced language features can blur this issue.

We use a standard definition of noninterference [26, 34]. Because we will only apply noninterference to straight-line programs the exact flavor (e.g., termination-sensitive vs. insensitive [50]) is inconsequential.

\textbf{2.2.1.1 Definition [Noninterference]}: A program $c$ satisfies noninterference, written $NI \models c$, iff whenever $m_1 =_L m_2$ and $(c, m_1) \xrightarrow{\ell_1}^* (\varepsilon, m'_1)$ and $(c, m_2) \xrightarrow{\ell_2}^* (\varepsilon, m'_2)$, we have $\ell_1 = \ell_2$.

Finally, we define weak secrecy in terms of the extracted explicit flow statements from all possible runs:

\textbf{2.2.1.2 Definition [Weak secrecy]}: A program $c$ satisfies weak secrecy, written $WS \models c$, iff whenever $(c, m) \xrightarrow{d}^* (c', m')$, we have $NI \models d$.

Consider the following program (call it $c_{impl}$):

if ($h = 0$)
    then 1 := 1
else 1 := 2;
out 1

0 otherwise.
This program leaks whether or not the high variable $h$ is 0 using an \textit{implicit flow}. For $c$ to satisfy weak secrecy, all possible sequences of assignment and output statements arising from executions of this program in different starting memories must be noninterfering. Concretely, $l := 1$ and $l := 2$, as well as $l := 1$; \textbf{out} $l$ and $l := 2$; \textbf{out} $l$ have to satisfy noninterference. Since this is the case, $WS \models c$ holds. This captures the intuition that taint is not propagated through control-flow decisions in taint-tracking systems.

Weak secrecy offers an intuitive formal account of security with respect to explicit flows. However, its inherently syntactic nature ("extract the assignment statements and outputs from every run of the program...") means that it is not always clear how to extend it to other programming languages. For example: (i) Many high-level languages allow the expressions appearing in the guards of conditionals and loops to have side effects. In such languages, the extracted statement will have to include more than just top-level assignment statements and outputs from the original program. E.g., for a program such as \textbf{if} $(h := 0) = 0$ \textbf{then} ... \textbf{else} ..., the assignment $h := 0$ has to be extracted. (ii) Conversely, many languages allow conditional \textit{expressions} such as $x ? y : z$ in C. If such expressions are regarded as “control flow,” then some significant transformation may be required to extract just the data-flow parts of expressions. Weak secrecy gives no direct guidance as to how this should be done.

One particular setting where taint tracking is commonly employed in practice is machine code. But adapting weak secrecy to machine code again raises questions: (i) Machine-code programs can access their own “program text” as data (and they sometimes do this, in practice, for example in certain tamper-proofing techniques [44]). If we extract only a part of this program text into the extracted program, then program memory will contain different values when the extracted program is executed, compared to the original. The extracted program can then produce different results, resulting in insecure flows that only occur in the extracted program but not in the original, or vice versa. We discuss this further in Section 2.2.4. (ii) Some instructions, e.g. \textit{jal} (jump and link), modify both the machine’s memory (the stack) and its control state (the program counter); that is, there is both an explicit as well as an implicit flow of information. To extract only the explicit
flow part, it seems jal instructions would have to be extracted into instructions capturing only the changes to the memory.

2.2.2 Explicit Secrecy

To deal uniformly with all these issues, we introduce explicit secrecy, a semantic property that builds on the intuition of weak secrecy. Intuitively, explicit secrecy separates changes to the state of a program from changes to the control flow, like weak secrecy. However, rather than manipulating the syntax of the running program, we now extract a function that captures the “state modification” of each execution step.

Explicit secrecy is language-agnostic, assuming only that the language designer can give a (small-step) semantics for their language or machine and specify the distinction between the “state” and “control” parts of a program or machine configuration with this choice determining what is considered to be an explicit flow. Roughly speaking, choosing a larger part of a configuration as the “state” part entails considering more flows as explicit flows. Concretely, the interface between the language definition and the explicit secrecy property comprises the following:

i. Sets of configurations (Conf), commands (Com), states (State), and observations (Obs). Intuitively, a state denotes only the “memory part” (as opposed to the “control part,” which is a command) of a configuration. An observation is an event that is visible to the attacker. (We sometimes say “program” instead of “command” when we want to emphasize that we are talking about an entire program such as might be loaded into a machine at the beginning of a run, as opposed to smaller fragments such as single machine instructions.)

ii. A small-step evaluation relation: \( \rightarrow \subseteq \text{Conf} \times \text{Obs}^* \times \text{Conf} \). We write \( \text{cfg} \xrightarrow{\alpha} \text{cfg}' \) if \( \text{cfg} \) evaluates to \( \text{cfg}' \) in one step while producing observations \( \alpha \); we elide \( \alpha \) when it is empty or unimportant. For simplicity, we assume that this relation is deterministic. (We discuss how to lift this assumption later in this section.)

iii. An equivalence relation \( \equiv_{\text{L}} \subseteq \text{State} \times \text{State} \) between states, where \( s_1 \equiv_{\text{L}} s_2 \) means that the states \( s_1 \) and \( s_2 \) are not
2.2. Specifying Explicit Flows

A function \( \text{state}(\cdot) : \text{Conf} \rightarrow \text{State} \) mapping configurations to states and a function \( \text{com}(\cdot) : \text{Conf} \rightarrow \text{Com} \) that extracts the next command that will be executed. Moreover, we require a function \( \langle \cdot, \cdot \rangle : \text{Com} \times \text{State} \rightarrow \text{Conf} \) constructing a configuration with a given command and state. For every configuration \( \text{cfg} \) it must hold that \( \langle \text{com}(\text{cfg}), \text{state}(\text{cfg}) \rangle = \text{cfg} \).

(In the instantiation for the while language in Section 2.2.4, configurations are simply tuples of command and state parts; \( \langle \cdot, \cdot \rangle \) constructs a tuple, with \( \text{state}(\cdot) \) and \( \text{com}(\cdot) \) acting as projections and obeying the usual laws. In the case of machine code, however, data and code are mixed. Therefore, the command that is extracted consists only of the next instruction, since other memory content cannot be distinguished from data that will not be executed. Thus, \( \text{com}(\langle c, s \rangle) = c \) does not hold in general for machine code. We discuss this in more detail in Section 2.2.4.)

The power of explicit secrecy depends on careful choice of the parameters. Intuitively, explicit flows are only concerned with changes to the state. To model explicit flows, we use the small-step semantics to construct, for each execution step during a program’s execution, a function that transforms a state while possibly producing output events. For simplicity, we add another assumption to guarantee totality of the function constructed for each step: we assume that, if \( \text{cfg} \rightarrow \text{cfg}' \), then for every configuration \( \text{cfg}_1 \) with \( \text{com}(\text{cfg}) = \text{com}(\text{cfg}_1) \), there exists \( \text{cfg}_1' \) such that \( \text{cfg}_1 \rightarrow \text{cfg}_1' \). I.e., evaluation of a fixed command is defined for all possible states, if that command can be evaluated in some state. This, together with assuming a deterministic evaluation relation, allows us to extract functions acting on state in a unique and well-defined way.

For example, if the entire configuration of a program is considered to be the state component, explicit secrecy coincides with (lock-step) noninterference. Similarly, for big-step semantics, there are no individual changes to state components in the semantics and explicit secrecy also coincides with noninterference. The force of the explicit secrecy policy depends on the choice of these parameters.

2.2.2.1 Definition: For each step \( \text{cfg} \rightarrow \text{cfg}' \), we define a function \( f : \text{State} \rightarrow \text{State} \times \text{Obs}^* \) by \( f(s) = (\text{state}(\text{cfg}''), \alpha) \) for the unique
\(\alpha, \text{cfg}''\) such that \(\langle \text{com}(\text{cfg}), s \rangle \xrightarrow{\alpha} \text{cfg}''\). We write \(\text{cfg} \xrightarrow{f} \text{cfg}'\) to denote that \(\text{cfg}\) evaluating to \(\text{cfg}'\) is associated with \(f\) in this way.

(Whether the subscript on \(\rightarrow\) refers to a function, or an extracted command in the sense of weak secrecy, should always be clear from context.)

Intuitively, for a step \(\text{cfg} \xrightarrow{f} \text{cfg}'\), \(f(s)\) simulates how the given input state \(s\) would be changed by executing the same step that \(\text{cfg}\) performs. For example, executing an assignment \(x := e\) in the while language results in \(f(m) = (m[x \mapsto m(e)], \varepsilon)\), where \(m(e)\) denotes evaluating expression \(e\) in memory \(m\); i.e., executing an assignment modifies the memory accordingly while producing no output. Performing a step for a command of the form \textbf{if} \(e\) \textbf{then} \(c_1\) \textbf{else} \(c_2\) results in \(f(m) = (m, \varepsilon)\), encoding the fact that branching neither changes the memory nor produces any output.

We can then lift the construction of state changing functions to multiple steps by chaining the state modifications and concatenating the produced output

\[
\text{cfg} \xrightarrow{id} \ast \text{cfg} \quad \frac{\text{cfg} \xrightarrow{\alpha} \ast \text{cfg}' \quad \text{cfg}' \xrightarrow{\beta} \text{cfg}''}{\text{cfg} \xrightarrow{\alpha \beta} \ast \text{cfg}''}
\]

where \(g \circ f\) computes the final state and concatenates the outputs—i.e., \((g \circ f)(s) = (s'', \alpha.\beta)\), when \(f(s) = (s', \alpha)\) and \(g(s') = (s'', \beta)\).

For example, consider the command \(c = (h := 0; \textbf{out} \ h)\) in the simple imperative language of Section 2.2.1. The function \(g\) associated with executing two steps of this program, i.e., \((c, m) \xrightarrow{\ell} \ast\)

\((\varepsilon, m')\), will first perform update the current memory and then output the value of \(h\) in the modified memory. That is, \(g = g_2 \circ g_1\) where \(g_1(m) = (m[h \mapsto 0], \varepsilon)\) and \(g_2(m) = (m, [m(h)])\), and hence \(g(m) = (m[h \mapsto 0], [0])\).

Functions constructed this way correctly encode actual execution steps:

\textbf{2.2.2.2 Lemma:} If \(\text{cfg} \xrightarrow{\alpha} \ast \text{cfg}'\), then

\(f(\text{state}(\text{cfg})) = (\text{state}(\text{cfg}'), \alpha)\).

Depending on how programs and configurations are represented, not all initial states might be valid starting states for a program.
For example, in the low-level language we consider, starting configurations are created from a program, represented as a list of instructions, and a starting memory by overwriting a part of the memory with the instructions of the program. Therefore, considering states where the program region contains different instructions can render well-behaved programs insecure. (A concrete example of this can be found in Section 2.2.4.) To rule out such “impossible” initial states, we define valid initial states as the set of states that can be obtained by creating a configuration from the command:

2.2.2.3 Definition [Initial states]: For command $c$, we define the set of valid initial states $\mathcal{S}_0(c)$ by

$$\mathcal{S}_0(c) = \{s \mid \exists s'. \text{state}(\langle c, s' \rangle) = s\}.$$  

In the case of machine code programs, this set will contain memories that match the instructions of $c$ at the addresses where $c$ is placed. In the case of while programs, all states are valid initial states.

We can now define the knowledge an attacker obtains from observing only outputs from a sequence of changes to the state. We capture this by defining a set of initial states that the attacker considers possible based on some observations. Concretely, for a given initial state $s_0$ and some state transformer $f$, another state is considered possible if $s_0 =_L s$ and it matches the trace produced by $f(s_0)$, i.e. $\pi_2(f(s_0)) = \pi_2(f(s))$, where $\pi_i$ projects a tuple to its $i$th component.

2.2.2.4 Definition [Explicit knowledge]: We define the explicit knowledge with respect to command $c$, initial state $s_0$, and state transformer $f$ by

$$k_e(c, s_0, f) = \{s \mid s =_L s_0 \land s \in \mathcal{S}_0(c) \land \pi_2(f(s)) = \pi_2(f(s_0))\}.$$  

A program then satisfies explicit secrecy for some initial state iff no indistinguishable, valid initial states can be ruled out from observing the output generated by the extracted state transformer.

2.2.2.5 Definition [Explicit secrecy]: A program $c$ satisfies explicit secrecy for initial state $s$, written $ES \models (c, s)$, iff whenever
⟨c, s⟩ \xrightarrow{f}^* cfg', we have ∀s_0 ∈ S_0(c). k_e(c, s_0, f_0) = k_e(c, s_0, f) where f_0(s) = (s, ε). A program c satisfies explicit secrecy, written $ES \models c$, iff $ES \models (c, s)$ for all $s \in State$.

In order to ignore information from implicit flows, the definition quantifies over all $s_0 ∈ S_0(c)$ instead of states that are low-equivalent to s. For example, consider the program if $l = 0$ then out $l \times h$ else skip, which is considered insecure by most taint-tracking systems. The state transformer extracted for the then branch (for a memory $m_0$ where $m_0(l) = 0$) is $f(m) = (m, m(l) \times m(h))$. If we consider only states that are low-equivalent to $m_0$, the explicit flow in the then branch is not detected, since low equivalence to $m_0$ implies that $l = 0$. In order to capture the behavior of taint-tracking systems, we quantify over all (valid) initial states instead.

Since the definition is knowledge-based, it extends naturally to a non-deterministic setting by extending state transformers to return sets of possible successor states. Similarly, the totality assumption on the evaluation relation can be lifted by constructing partial state transformers and considering only states in the domain of the resulting state transformers.

### 2.2.3 Declassification

Like noninterference, explicit secrecy is often too strict to accommodate real-world applications and needs a mechanism to declassify some data that depends on sensitive information. For example, taint-tracking systems often allow taint to be removed from data after passing it through a sanitizer. In this section we show how explicit secrecy can be extended to handle such behavior.

Being a knowledge-based definition, explicit secrecy can be extended naturally to support declassification in the style of gradual release [11]. We assume that there is a set of release events $Rel \subseteq Obs$ that occur when information is intentionally released by a program. In terms of explicit secrecy, we allow the attacker’s knowledge to change based on observing such events but require that it remain constant for other events.

#### 2.2.3.1 Definition [Explicit secrecy modulo release]

A command c then satisfies explicit secrecy modulo release for initial state
For example, the while language from Section 2.2.1 can be extended with a `declassify(e)` statement that releases the value of expression `e` to the attacker. The semantics of the language is extended by the following rule:

\[
E-\text{Decl} \\
\frac{m(e) = n}{(\text{declassify}(e), m) \xrightarrow{\text{rel}(n)} (\varepsilon, m)}
\]

The rule denotes that evaluating a statement `declassify(e)` produces a release event `\text{rel}(n)` containing value `n`. We then define the set of release events as \( \text{Rel} = \{ \text{rel}(n) \mid n \in \mathbb{Z} \} \).

To illustrate this extension, consider a program that processes some sensitive input (such as a POST request containing a password and other data) and logs information about requests to a log file. It is important that the user’s password not be leaked to the log file. For example, the following program logs the request verbatim, thereby violating the intended policy (this is often mentioned as a secure coding guideline as well [2]):

```plaintext
output_to_log ("Request: " + request);
if (hash(password(request)) == stored_hash)
then output_to_user sensitive_data
else output_to_user "access denied";
```

(Here `password` extracts the submitted password and `hash` computes the hash of a password.) This program is rejected by explicit secrecy since private information, namely the contents of `request`, are output explicitly (to the log file).

If instead the request is sanitized by removing the password, the program is secure and it is accepted:

```plaintext
declassify(remove_password(request));
output_to_log ("Request: " + remove_password(request));
if (hash(password(request)) == stored_hash)
then output_to_user sensitive_data
else output_to_user "access denied";
```
Because of the use of declassification, this program satisfies explicit secrecy modulo release: The attacker learns the value of `remove_password(request)` when the `declassify` command is executed; this is permitted, since the produced event is a release event. The same value is then appended to the log, resulting in no increase in knowledge for the attacker. (The final output statement also depends on the user’s password, but does so only through the control flow and hence is accepted by explicit secrecy.)

### 2.2.4 Instantiating Explicit Secrecy

This section demonstrates how to instantiate the explicit secrecy framework both for the high-level language introduced in Section 2.2.1 and for a RISC-style assembly language.

As described in Section 2.2.2, instantiating explicit secrecy first requires the language designer to define which parts of a configuration hold state and which determine the control flow of a program. This is specified by providing mapping functions from configurations to state and command parts respectively.

For example, in many imperative high-level languages, configurations often consist of the current statement, along with various forms of state, such as values of variables, the state of the heap and stack, exception handlers, etc. In such a case, the statement and exception handlers (since they are part of the control flow) can be considered the control-flow component and the variables and the heap and stack as the state component of a configuration. Also note that, in the extremes, choosing the entire configuration as control-flow component yields a vacuously true condition, while taking the entire configuration to be the state part results in a form of noninterference. For example, in a purely functional language without side effects, the entire configuration has to be considered as either just state or just control, showing that taint tracking for (purely) functional languages is not a very meaningful notion.

Second, explicit secrecy assumes a small-step evaluation relation provided by the language designer, which also determines the events that are observable by the attacker. To model a powerful attacker observing all writes to public memory locations, each step of an assignment statement to a low variable would produce an attacker observation. Conversely, for an attacker who only observes
specific output events, observations are only produced when evaluating distinguished output statements.

To summarize, the following steps are needed when instantiating explicit secrecy for a new language:

- Decide what part of a configuration contains data. Explicit secrecy then only tracks statements that directly modify this part. Changes to other parts of the configuration are not considered for security. A useful rule of thumb may be to choose all parts of a configuration that directly contain secrets. Additionally, the language designer has to specify a function constructing configurations from a command and a state part.

- Choose an appropriate small-step relation. In order to arrive at a sensible security condition, evaluation steps need to be small enough to separate changes to the state from changes to the control-flow. For example, a big-step relation results in a much stronger security condition, similar to noninterference.

- Choose the attacker observations. This depends on the application scenario and affects how strong the resulting security condition is.

- Depending on the attacker, one should also choose an equivalence relation on states encoding what the attacker can observe about initial states.

**Instantiation for While Language**

To instantiate explicit secrecy for the language introduced in Section 2.2.1, we define the parametric elements from Section 2.2.2 as follows:

i. The set of states is given by memories, i.e., functions mapping variables to values \( \text{State} = \text{Mem} = \text{Var} \rightarrow \mathbb{Z} \), and \( \text{Com} \) is the set of statements defined in Section 2.2.1. A configuration is a pair of a command and a memory: \( \text{Conf} = \text{Com} \times \text{Mem} \). An observation is a value \( v \in \mathbb{Z} \).

ii. The evaluation relation \( \rightarrow \) is as given in Section 2.2.1.
iii. As before, $m_1 \equiv m_2$ holds iff $\forall x. \Gamma(x) = L \Rightarrow m_1(x) = m_2(x)$.

iv. Projecting to the state or command component of a configuration extracts the memory $\text{state}((c, m)) = m$ or command part $\text{com}((c, m)) = c$; creating a configuration from a given command and memory is given by $\langle c, m \rangle = (c, m)$.

Each step results in a state transformer based on the first statement in the configuration. The state transformer acts on the memory part of the configuration, and might produce an output: An assignment $x := e$ will yield a function $f(m) = (m[x \mapsto m(e)], \varepsilon)$. Output statements of the form $\text{out } e$ result in a function $f(m) = (m, [m(e)])$. For every other type of statement, the extracted function is $f(m) = (m, \varepsilon)$. In particular this means that control-flow commands such as if statements will not affect the attacker’s knowledge, corresponding to the intuition that these are not considered explicit flows.

As an example, consider the program $c_{\text{impl}}$ from Section 2.2.1, which leaks one bit about the value of $h$ using an implicit flow. The possible state transformers $f$ for a complete execution of $c_{\text{impl}}$ starting in $m_0$ have the form $f_i(m) = (m[l \mapsto i], [i])$ where $i = 1$ or $i = 2$, depending on whether or not $m_0(h) = 0$. If we consider any other memory $m$ such that $m \in [s_0]L$, then it holds that $\pi_2(f_i(m)) = [i] = \pi_2(f_i(m_0))$. Therefore, the program is judged secure by explicit secrecy, since no information is leaked through an explicit flow.

However, a program such as if $h = 0$ then $\text{out } h$ else $\text{out } 0$ will be judged as insecure since, if $m_0(h) = 0$, the extracted state transformer is $f(m) = (m, [m(h)])$, due to the elimination of control-flow information. Even though this program is secure in the sense of noninterference, the program is not secure in the sense of explicit secrecy, since information is propagated using an explicit flow.

After performing a series of steps $(c, m) \xrightarrow{f^*} (c', m')$, the function $f$ that is extracted from the run corresponds to the sequence of assignments and output statements executed; since the steps that only alter the control flow, such as if statements, do not change the memory, these statements are not reflected in $f$. This corresponds to the sequence of commands that is extracted by Weak Secrecy and in fact, weak secrecy and explicit secrecy for while programs
2.2. Specifying Explicit Flows

coincide:

2.2.4.1 Theorem: ES ⊨ c iff WS ⊨ c, for any command c.

A natural question that arises is how explicit secrecy applies to languages with richer features, such as pointers or reflection. For example, in a language supporting pointers, the extracted state transformers, depending on the exact semantics, would replicate the changes pointer-based features make to the memory. A statement like *e₁ := e₂ in a C-like language assigning the result of e₂ to the location pointed to by e₁ would result in the expected state transformer f: Namely f(m) would evaluate e₁ in m, and assign the result of evaluating e₂ in m to that location and return the modified memory. This captures the state modification performed by the original statement. This would imply that an enforcement mechanism which taints all variables occurring in e₁ would be sound, provided the rest of the language features are handled properly. This is discussed in more detail in the context of machine code in the following section.

Similarly, features like Java-style reflection would result in state transformers capturing the modifications they perform to the stack and heap, but ignore implicit flows resulting from it; for example code constructed at runtime.

Instantiation for Machine Code

We next demonstrate how to instantiate explicit secrecy for a simple RISC-style machine. We denote the set of machine words by \( \mathbb{W} \) and the set of registers by \( \text{Reg} \). We consider the following instruction set:

\[
i ::= \text{nop} \mid \text{halt} \mid \text{out} \ r \mid \text{const} \ i \ r \mid \text{mov} \ r_{\text{src}} \ r_{\text{dst}} \mid \text{op}_\oplus \ r_1 \ r_2 \ r_{\text{dst}} \mid \text{load} \ r_{\text{addr}} \ r_{\text{dst}} \mid \text{store} \ r_{\text{addr}} \ r_{\text{val}} \mid \text{jump} \ r \mid \text{bnz} \ r \ i \mid \text{jal} \ r
\]

The semantics of instructions are standard (full details are in Appendix 2.B.): \text{nop} performs no operation, \text{halt} halts execution, and \text{out} \ r outputs the content of register \( r \). Instructions of the form \text{const} \ i \ r load constant \( i \in \mathbb{W} \) into register \( r \), \text{op}_\oplus \ r_1 \ r_2 \ r_{\text{dst}} \) combines registers \( r_1 \) and \( r_2 \) using operator \( \oplus \) and stores the result in register \( r_{\text{dst}} \). Memory access is performed using \text{load} and \text{store}
instructions where load $r_{addr}$ $r_{dst}$ loads from the address in $r_{addr}$ into $r_{dst}$ and store $r_{addr}$ $r_{val}$ writes the value in $r_{val}$ to the memory address stored in $r_{addr}$. The control flow can be manipulated using jump $r$, which jumps to the address in register $r$, or bnz $r$ $i$, which jumps to address $i$ if the value in $r$ is non-zero. Function calls can be performed using an instruction of the form jal $r$ which stores the current program counter at the address pointed to by a special register $sp$ and jumps to the address in register $r$.

A configuration $(\text{mem}, \text{reg}, \text{pc})$ of this machine consists of a memory state $\text{mem} : \mathbb{W} \rightarrow \mathbb{W}$, a register state $\text{reg} : \text{Reg} \rightarrow \mathbb{W}$, and a program counter $\text{pc} \in \mathbb{W}$. Evaluation of configuration $(\text{mem}, \text{reg}, \text{pc})$ in one step to configuration $(\text{mem}', \text{reg}', \text{pc}')$ while producing trace $\alpha \in \mathbb{W}^*$ is denoted by $(\text{mem}, \text{reg}, \text{pc}) \xrightarrow{\alpha} (\text{mem}', \text{reg}', \text{pc}')$. Machine code instructions are also encoded as machine words; decode$(w)$ turns a machine word into a symbolic representation of the instruction (if possible), while encode$(i)$ maps instruction $i$ to the corresponding machine word. A program $(\text{is}, \text{pc}_0)$ for this machine consists of a list of machine words $\text{is} \in \mathbb{W}^*$ together with a word indicating the expected starting address. A starting machine configuration for $(\text{is}, \text{pc}_0)$ is produced from an initial memory $\text{mem}$ and initial register state $\text{reg}$ by overriding the words at $\text{pc}_0, \ldots, \text{pc}_0 + |\text{is}| - 1$ by the instructions $\text{is}$ and setting the program counter to $\text{pc}_0$. We denote this by $(\text{mem}[\text{pc}_0 \mapsto \text{is}], \text{reg}, \text{pc}_0)$.

We assume that each memory location $a \in \mathbb{W}$ has an associated security level $\Gamma(a) \in \mathcal{L}$; similarly, each register $r \in \text{Reg}$ is associated with a security level $\Gamma(r) \in \mathcal{L}$.

We instantiate the parameters from Section 2.2.2 as follows:

i. States consist of a function mapping addresses, represented as machine words, to words and a function mapping registers to words: $\text{State} = (\mathbb{W} \rightarrow \mathbb{W}) \times (\text{Reg} \rightarrow \mathbb{W})$. The commands are given by a sequence of instructions together with a starting address $\text{Com} = \mathbb{W}^* \times \mathbb{W}$. A configuration $(\text{mem}, \text{reg}, \text{pc})$ is a triple, consisting of a memory state $\text{mem}$, a register state $\text{reg}$, and a program counter $\text{pc}$. I.e., $\text{Conf} = (\mathbb{W} \rightarrow \mathbb{W}) \times (\text{Reg} \rightarrow \mathbb{W}) \times \mathbb{W}$.

ii. The evaluation relation $\xrightarrow{}$ is standard (details in Appendix 2.B).

iii. $(\text{mem}_1, \text{reg}_1) =_L (\text{mem}_2, \text{reg}_2)$ holds iff $\forall a. \Gamma(a) = L \Rightarrow \text{mem}_1(a) = \text{mem}_2(a)$ and $\forall r. \Gamma(r) = L \Rightarrow \text{reg}_1(r) = \text{reg}_2(r)$. 


iv. Extracting the state of a configuration returns the memory and registers: \( \text{state}((\text{mem}, \text{reg}, \text{pc})) = (\text{mem}, \text{reg}) \). Extracting the next instruction returns the current instruction and the value of the program counter: \( \text{com}((\text{mem}, \text{reg}, \text{pc})) = (\text{mem}[\text{pc}], \text{pc}) \). Note that this does not extract the entire program; only the next instruction to be executed is returned.

Constructing a new configuration is defined by
\[
((\text{is}, \text{pc}_0), (\text{mem}, \text{reg})) = (\text{mem}[\text{pc}_0 \mapsto \text{is}], \text{reg}, \text{pc}_0)
\]
where \( \text{mem}[\text{pc}_0 \mapsto \text{is}] \) denotes replacing the words at \( \text{pc}_0, \text{pc}_0 + 1, \ldots, \text{pc}_0 + |\text{is}| \) by \( \text{is}_0, \text{is}_1, \ldots, \text{is}_{|\text{is}|} \).

The extracted functions model the effect the various instructions have on the memory and registers. For example, the instruction \texttt{mov} \( r_s \ r_d \) results in \( f((\text{mem}, \text{reg})) = ((\text{mem}, \text{reg}[\text{reg}[r_s]] \mapsto \text{reg}[r_d]), \varepsilon) \), while \texttt{store} \( r_a \ r_v \) generates the state transformer \( f((\text{mem}, \text{reg})) = ((\text{mem}[\text{reg}[r_a] \mapsto \text{reg}[r_v]], \text{reg}), \varepsilon) \). Performing an output instruction \texttt{out} \( r \) induces the function \( f((\text{mem}, \text{reg})) = ((\text{mem}, \text{reg}), \text{reg}[r]) \).

Note that mixing instructions and memory to create a configuration rules out some invalid initial states: For example, consider the following program that outputs the value of \( (\text{mem}[5] - x) \times \text{mem}[h] \), for some constant \( x \).

```plaintext
1 const 5 r1  // load constant 5 into r1
2 load r1 r2  // load mem[5] into r2
3 const x r1  // load constant x into r1
4 sub r2 r1 r2 // r2 = mem[5] - x
5 const h r1  // load constant h into r1
6 load r1 r3  // load mem[h] into r3
7 mul r2 r3 r2 // r2 = (mem[5] - x) * mem[h]
8 out r2      // output (mem[5] - x) * mem[h]
```

In the case where \( x = \text{encode} \texttt{(load} r_1 r_3) \), the program always produces the trace \([0]\), since this instruction will always be inserted at address \( 6 \) when creating a starting configuration. The function that is extracted from a run of this program is
\[
f((\text{mem}, \text{reg})) = ((\text{mem}, \text{reg}), (\text{mem}[6] - x) \times \text{mem}[h])
\]
which will produce a trace different from \([0]\) if \( \text{mem}[6] \neq \text{encode} \texttt{(load} r_1 r_3) \). However, such a state does not correspond to a valid execution of the program and should not be considered when judging its security. This is addressed by the notion of
valid initial states from Section 2.2.2. For any machine-code program \((is_0 \ldots is_n, pc_0)\), it holds that \(S_0((is, pc_0)) = \{(mem, reg) \mid mem[pc_0] = is_0, \ldots mem[pc_0 + |is| - 1] = is_n\}\), thereby ruling out impossible starting states when determining whether or not a program satisfies explicit secrecy.

Recall the key intricacy connected to applying weak secrecy to low-level programs: programs exist in the memory and their instructions can then be read and used for computations. As a simplified example, on how this can affect a security analysis, consider the following program:

```plaintext
1  const 3  r1  // put 3 into r1
2  load r1 r2  // load from mem[r1] into r2
3  bnz 17 r2  // assume a non-zero opcode
4  const r1 x  // load constant x into r1
5  sub r2 r1 r2  // set r2 := mem[3] - x
6  const h r1  // load address h into r1
7  load r4 r1  // load mem[r4] into r1
8  mul r1 r4 r2  // set r2 := (mem[3] - x) * mem[h]
9  out r2  // output (mem[3] - x) * mem[h]
```

This program reads the opcode of instruction 3, i.e., loads the value \(encode(bnz 17 r_2)\) into register \(r_2\), subtracts a constant \(x\) and outputs the result of multiplying \(r_2\) with a value from a high memory location \(h\). If we set \(x = encode(bnz 17 r_2)\), then every run of this program will produce \([0]\). An adaptation of weak secrecy to the machine code would involve modifying the executed sequence of instructions by eliminating branching instructions such as \(bnz 17 r_2\). But such a modification might well result in a different value than \(encode(bnz 17 r_2)\) being loaded into register \(r_1\) in the second instruction, resulting in a trace that depends on the value of \(h\) and renders the program insecure. In contrast, explicit secrecy handles such programs seamlessly since the same memory as in the original execution is used.

The instantiation of explicit secrecy also provides a starting point on how to suitably extend weak secrecy: instead of replacing part of the memory and using the same machine to execute the extracted program, the machine can be modified to read instructions from another part of the configuration while acting on the same memory as the original program. I.e., the original program is placed in the starting memory for an alternate machine that reads instructions from a list instead of the memory. We then check the extract instructions for noninterference with re-
2.2. Specifying Explicit Flows

spect to this modified machine. Moreover, note that an instruction of the form jal $r$ leads to a state transformer of the form $f((\text{mem}, \text{reg})) = ((\text{mem}[\text{reg}[sp] \mapsto pc], \text{reg}), \varepsilon)$ where $pc$ is the program counter when the jal instruction gets executed. This also leads to an intuition of how to encode capture state modification of jal instruction: when encountering such an instruction in a configuration with program counter $pc$, we extract the instructions storing the value $pc$ at the address pointed to by $sp$. We present this extension of weak secrecy to low-level machines in the Appendix and show that it coincides with explicit secrecy.

2.2.5 Explicit Secrecy in the Big Picture

We close the discussion of soundness conditions for taint tracking by examining the relation between explicit secrecy and standard notions from the area of information-flow control. We present an intuitive property aimed at capturing explicit flows as a variant of gradual release, a knowledge-based approach to noninterference with a declassification policy, and we review how explicit secrecy relates to noninterference. We discuss this approach in the context of the while language from Section 2.2.1, but it should generalize straightforwardly to other settings, such as machine code.

Intuitively, allowing information leakage due to implicit flows can be thought of as releasing (or declassifying) information about control-flow decisions taken during the run of a program; i.e., any high information used to determine control flow is considered to be disclosed to the attacker, but they should learn nothing else. To accommodate this form of declassification, we extend traces with release events of the form $\text{rel}(v)$ where $v \in \mathbb{Z}$. Every control-flow decision then generates an event of this form to release information about implicit flows to the attacker. Concretely, we modify the rules for control-flow commands in the semantics to produce release events. Below we show the rules for the “true” case for $\text{if}$ and $\text{while}$ commands (the other cases produce similar events).

\[
\begin{align*}
\text{if } e \text{ then } c_1 \text{ else } c_2, m &\xrightarrow{\text{rel}(n)} (c_1, m) \\
\text{while } e \text{ do } c, m &\xrightarrow{\text{rel}(n)} (c; \text{while } e \text{ do } c, m)
\end{align*}
\]
The intuition is then to allow the attacker’s knowledge to increase based on release events, but not at any other point during the execution. We first define the knowledge of the attacker in the standard way [11]:

**2.2.5.1 Definition [Knowledge]**: The knowledge set for statement $c$, initial memory $m_0$, and trace $\ell$ is given by $k(c, m_0, \ell) = \{m \mid m \models \mathbb{L} m_0 \land (\exists c', m'. (c, m) \xrightarrow{\ell}^* (c', m')))\}$.

Based on this, we introduce *control-flow gradual release*, specifying that changes in knowledge are only allowed due to release events:

**2.2.5.2 Definition [Control-flow gradual release]**: A program $c$ satisfies control-flow gradual release, written $\text{CFGR} \models c$, iff whenever $(c, m) \xrightarrow{\ell, \alpha}^* (c', m')$ and $\forall n. \alpha \neq r(n)$, we have $k(c, m, \ell) = k(c, m, \ell, \alpha)$.

As an example, consider again the program $c_{\text{impl}}$ from Section 2.2.1, executed with some initial memory $m_0$ where $m_0(h) = 0$. Initially, $k(c_{\text{impl}}, m_0, \varepsilon) = [m_0]_L$. After performing the branching for $\text{if } h = 0$, the release event $r(1)$ is produced, changing the knowledge set to $k(c_{\text{impl}}, m_0, [r(1)]) = \{m \mid m \models \mathbb{L} m_0 \land m(h) = 0\}$. Performing the output $\text{out } 1$ produces the event 1. However, the knowledge set remains unchanged: $k(c_{\text{impl}}, m_0, [r(1), 1]) = k(c_{\text{impl}}, m_0, [r(1)])$. The knowledge changes in an analogous way if $m_0(h) \neq 0$.

Control-flow gradual release is more permissive than explicit secrecy:

**2.2.5.3 Theorem**: If $\text{ES} \models c$, then $\text{CFGR} \models c$.

Perhaps surprisingly, this inclusion is strict: there are programs satisfying control-flow gradual release but not explicit secrecy. For example, consider a program that leaks the same information twice, first using an implicit and then an explicit flow:

```plaintext
if (h = 0)
    then out 1
    else out 2;
out (h = 0)
```
Upon reaching \textbf{out} \((h = 0)\), the knowledge set is already reduced to memories \(m\) where \(m(h = 0) = m_0(h = 0)\). Thus, seeing the last output does not affect the knowledge of the attacker.

This program, however, is not accepted by explicit secrecy, as the extracted functions have the form \(f(m) = (m, [i, m(h = 0)])\) for either \(i = 1\) or \(i = 2\), which produce different traces depending on whether or not \(h = 0\). Practical taint-tracking systems would also reject this program.

What about the relation to ordinary noninterference? As Volpano pointed out, weak secrecy (and therefore also explicit secrecy), despite its name, is in fact \emph{not} weaker than noninterference. For example, consider the following program \((c_{ni})\):

\begin{verbatim}
if (h = 0)
  then out h
else out 0
\end{verbatim}

In both branches, this program will produce the trace \([0]\). However, one sequence of outputs and assignments that can be extracted is \textbf{out} \(h\) and hence \(WS \not\models c_{ni}\).

In the case of straight-line programs, noninterference and weak secrecy do coincide. In particular this entails that weak secrecy and explicit secrecy, like noninterference, cannot be expressed as a property on traces [43]: both conditions are hyperproperties [24].

Since control-flow gradual release relaxes noninterference using a fixed declassification policy, the former is weaker than the latter:

\textbf{2.2.5.4 Theorem:} If \(NI \models c\), then \(CFGR \models c\).

The relations between the various properties can be summarized as follows. (We will see the enforcement mechanisms in Section 2.3.)
2.3 Enforcement

A large body of research literature shows that static and dynamic taint tracking can mitigate a wide range of confidentiality and integrity vulnerabilities. Applications of taint analysis cover the full range of the hardware and software stack, including memory safety violations such as buffer overruns in low-level code, injection attacks such as cross-site scripting in web applications, and recently also privacy leaks in smart phone apps. On the practical side, numerous existing tools and languages substantiate the applicability of taint tracking in these settings. For instance, binary analysis platforms such as BAP [18] and BitBlaze [55] implement taint tracking for analyzing native applications. Languages such as Perl [4] and Ruby [3] have built-in support for taint analysis, while language extensions such as Andromeda [59] and Pixy [38] support tainting in web applications. In the mobile context, TaintDroid [33] and FlowDroid [7] leverage taint tracking to enforce security for phone apps.

Despite the widespread usage of taint tracking, there has been little effort to formally define policies and mechanisms implemented by these tools. Notably, Schwartz et al. [53] and Livshits [41] formalize the essence of dynamic taint tracking by instrumenting the operational semantics of a core language for assembly and Java programs, respectively. Recent work by Bodin et al. [17] presents a technique for deriving semantic dependency analyses from a natural semantics of an imperative language with objects, which stands at the core of taint analysis. While this line of works provides useful insight for implementing correct taint analysis, it does not formally justify soundness for a security condition like weak secrecy. Such a condition however can help to understand what policies taint tracking ensures.

Our goal in this section is to check that weak secrecy and explicit secrecy correctly capture the intuition of existing taint-tracking systems. We prove soundness of flow-sensitive dynamic taint tracking for high-level code (Section 2.3.1) and machine code (Section 2.3.2) for explicit secrecy. Further, we present a static analysis for Java-like code based on symbolic execution and automated theorem proving and prove it sound for explicit secrecy (Section 2.3.3).
2.3.1 Dynamic Tainting for Imperative Code

We first discuss taint tracking in the context of the while language introduced in Section 2.2.1 and prove it sound with respect to the instantiation of explicit secrecy as described in Section 2.2.4.

A flow-sensitive dynamic taint tracker, as in real-world languages such as Perl or Ruby, keeps track of what variables have tainted content at each point during execution. To formalize this, we extend configurations with a function $\tau : \text{Var} \rightarrow \mathcal{L}$ mapping program variables to security levels. We replace the evaluation rules for assignments and output as follows (the other rules simply propagate the taints unchanged, as illustrated by T-IfTrue). As before, initial security labeling of program inputs, i.e., memories, is determined by $\Gamma$ and we write $\tau_0$ for $\Gamma$.

![Figure 2.1: Dynamic Tainting for Imperative Code](image)

Figure 2.1: Dynamic Tainting for Imperative Code
When evaluating an assignment $x := e$, the taint state $\tau(x)$ of variable $x$ is updated to match the taint of the expression that is being assigned to $x$. We extend $\tau$ to arbitrary expressions by defining $\tau(e) = \bigcup_{x \in \text{vars}(e)} \tau(x)$ where $\text{vars}(e)$ denotes the set of variables appearing (syntactically) in $e$. If a tainted variable occurs in an expression that is output, then the execution is stopped (written $\$\$). (Some tainting tools [46] use more precise analyses that can avoid fake dependencies such as $l := h - h$.) We can now show that whenever the dynamic taint tracker reaches a configuration $(c', m')$ starting from an initial configuration $(c, m)$, the program $c$ satisfies explicit secrecy with respect to that run.

2.3.1.1 Theorem [Soundness of Tainting]: For any program $c$, initial state $m$, and initial security labeling $\tau_0$, if $(c, m, \tau_0) \not\Rightarrow^* \$$, then $ES \models (c, m)$.

2.3.2 Dynamic Tainting for Machine Code

We now show how to enforce explicit secrecy by a flow-sensitive dynamic taint tracker for the machine language presented in Section 2.2.4. Analogously to taint tracking for imperative programs, we extend machine configurations with another component keeping track of which memory addresses and registers contain tainted data. The taint state is represented by a function $\tau : \text{Reg} \cup \mathbb{W} \to \mathcal{L}$ which maps registers and memory addresses to security levels. This function records the taint status for all registers and memory addresses, and uses them to derive the taint status for all values during the execution. Figure 2.2 presents the instrumented semantics rules for a dynamic taint tracker that handles self-modifying machine code. Our formalization resembles the ones used by real-world dynamic taint trackers such as those implemented by the CMU Binary Analysis Platform (BAP) [18] and the BitBlaze binary analysis platform [55]; it can also cope with self-modifying code (which they cannot). Schwartz et al. [53] formalize dynamic taint analysis by means of operational semantics for a low level language used in BAP and BitBlaze. Most of the challenges involve soundness of the taint-tracking mechanisms, especially undertainting. Explicit secrecy allows deciding which design choices result in unsoundness, making it easier to show whether this is appropriate in a particular
scenario. E.g., time-of-detection vs. time-of-attack issues can be addressed by appropriately choosing the attacker observations.

The taint state $\tau$ is updated as for the imperative programs. If a tainted register is used to output a value (rule $T\text{-OutFail}$), the execution is stopped. Otherwise the program performs the output and the execution proceeds (rule $T\text{-OutL}$). Rule $T\text{-Jal}$ updates the taint state by applying the join operator to combine the taint status of the address pointed by register $sp$ with the taint status of the current program counter $pc$. Similarly, rule $T\text{-Store}$ combines the taint status of the value to be stored with the taint status of the address that value is being stored to. Rule $T\text{-Load}$ combines the taint status of address it loads from with the taint status of target register and assigns the resulting label to that register, while rule $T\text{-Const}$ updates the taint status of the register as untainted.

For example, consider the following machine-code program with initial security labeling $\Gamma(r1) = \Gamma(1) = H$. The program loads a value from a tainted address and outputs its content to the attacker, and it is clearly insecure. In fact, this program does not satisfy explicit secrecy.

```plaintext
const 1 r1 // load constant 1 into r1
load r1 r1 // load mem[1] into r1
out r1 // output mem[1]
```

Now suppose we execute the program with $\tau_0 = \Gamma$ as the initial taints. Rule $T\text{-Const}$ updates the taint status of register $r1$ to $L$, hence $\tau_1 = \tau_0[r1 \rightarrow L]$. The successive load instruction will combine the taint status of register $r1$ with the taint status of address 1 as in rule $T\text{-Load}$, i.e., $\tau_2(r1) = \tau_1(r1) \sqcup \tau_1(1) = H$. Finally, rule $T\text{-OutFAIL}$ will stop the execution as expected, since $\tau_2(r1) = H$.

The following theorem shows that whenever the instrumented semantics in Figure 2.2 is accepted by taint tracking (i.e., it does not fail because of a violation of the taint policy), the original program satisfies explicit secrecy.

2.3.2.1 Theorem [Soundness of Tainting]: For any program $c = (is, pc_0)$, initial state $s_0 = (mem_0, reg_0)$ and initial security labeling $\tau_0$, if $(mem_0[pc_0 \mapsto is], reg_0, pc_0, \tau) \not\rightarrow^* \not\infty$ then $ES \models (c, s_0)$. 
2.3.3 Static Analysis for Taint Tracking

Lastly, we present a static analysis for enforcing explicit secrecy for imperative programs with a heap. The analysis leverages symbolic execution and theorem proving to statically certify the security of simple Java-like programs with respect to explicit secrecy. Static analysis stands at the core of several taint tracking tools. For instance, Andromeda and FlowDroid perform static taint analysis for web and mobile applications. While Andromeda’s and FlowDroid’s static analysis target the more challenging setting of real Java code, we extend our simple imperative language from Section 2.2.1 with a heap to provide a static analysis that checks for explicit secrecy. This new language is the same as the one used to describe the essence of static taint analysis implemented by the tools above [59, 7].

The language from Section 2.2.1 is extended with heap locations and fields, object creation, and load and store expressions on object fields.

\[ c := \cdots | x := \textbf{new} \ Object() | x := y.f | x.f := y \]

We extend the set of values \( \text{Val} \) with a set of object locations \( \text{Loc} \) and a null value. A configuration \((c, m, h)\) consists of a program \(c\), a memory \(m \in \text{Mem}\) and a heap \(h \in \text{Heap}\) mapping locations and field identifiers to values, i.e. \(\text{Heap} = \text{Loc} \times \text{Fld} \rightarrow \text{Val}\). The evaluation relation is extended as expected; details can be found in Appendix 2.A.1.

We reason about the behavior of a Java-like program by means of forward symbolic execution. Symbolic execution allows us to build a logical formula, which corresponds to a set of concrete program executions, and use first-order reasoning to statically prove whether or not that program satisfies explicit secrecy. We assume that there is a sound (but not necessarily complete) procedure for determining validity of first-order formulas. Concretely, such a procedure could be implemented by an SMT solver. We denote that a formula \(\varphi\) is reported as valid by SMT \(\vdash \varphi\). The program is executed on symbolic inputs, hence the state and the configuration are also symbolic.

A symbolic configuration \((c, \delta, \varphi)\) consists of a command \(c\), a symbolic state \(\delta\), and a path condition \(\varphi\), where \(\varphi\) is a (quantifier-free) first-order formula. A symbolic state \(\delta : (\text{Var} \rightarrow \text{Expr}) \cup\)
(\text{Loc} \times \text{Fld} \rightarrow \text{Expr}) \text{ is a mapping from program variables and object fields to expressions. The intuition is that } \delta(x) \text{ expresses the value of a variable } x \text{ or an object field } y.f \text{ at some point of the execution in terms of the values of initial variables. The symbolic state is updated when processing assignments to reflect changes to variables or fields. For example, after performing the assignments } x := y ; x := x + z, \text{ the symbolic state records the value of } x \text{ as } \delta(x) = y + z. \text{ Similarly, the program } x := \textbf{new} \text{ Object()} ; x.f := y \text{ yields a symbolic state } \delta \text{ such that } \delta(x) = l, \text{ for some fresh location } l \in \text{Loc}, \text{ and } \delta(l, f) = y. \text{ A path condition } \varphi \text{ is a symbolic boolean expression built over the initial variables and it constrains the set of concrete initial states to those that execute a given program path.}

Figure 2.3 presents the symbolic evaluation rules for statically checking explicit secrecy for the Java-like language. We denote a symbolic expression } e \text{ where all high (tainted) variables have been renamed (deterministically) to fresh variables by } \tilde{e}. \text{ Consider for example a boolean expression } e = (l + h > 0) \text{ such that } \Gamma(l) = \text{L} \text{ and } \Gamma(h) = \text{H}. \text{ Then } \tilde{e} = (l + h') \text{ for some fresh variable } h'. \text{ This simulates a second run with a low-equivalent memory in the style of self-composition [14]. Output instructions } \textbf{out} e \text{ are validated by checking for equivalence of the expression in the two memories by requiring that } \delta(e) = \tilde{\delta}(e). \text{ The merging of two symbolic states } \delta_1 \text{ and } \delta_2 \text{ when branching on expression } e \text{ is defined by } (\delta_1 \oplus_e \delta_2)(x) = \delta_1(x) \text{ if } \delta_1(x) = \delta_2(x) \text{ and } (\delta_1 \oplus_e \delta_2)(x) = (e ? \delta_1(x) : \delta_2(x)) \text{ if } \delta_1(x) \neq \delta_2(x). \text{ The merging operation allows for a compact representation of symbolic state modifications occurring in each branch: Instead of tracking modifications for each branch separately, we encode changes that occurred in each branch as one expression. For example, after the command } \textbf{if} e \textbf{ then} x := 1 \textbf{ else} x := 2, \text{ the variable contains either 1 or 2 depending on the branch that was taken. To avoid exploring the rest of the program for both possible values of } x, \text{ we encode this dependency as } \delta(x) = (e ? 1 : 2). \text{ At the same time, the branch condition is only needed to check whether branches are reachable. This analysis benefits from recording precise dependency information about } x \text{ instead of just tracking just 1 and 2 as possible values.}

In order to simulate executions taking the same branches, control-flow information has to be removed from } \delta. \text{ We introduce a function } \textbf{forget}(\cdot) : \text{Expr} \rightarrow \text{Expr}, \text{ where } \textbf{forget}(e) \text{ removes information re-}
resulting from merging of symbolic states after if statements. For example, consider the implicit flow example, cimpl. The dependency recorded for \( l \) after performing the assignment in the branch is \( \delta(l) = (h ? 1 : 2) \). Since this expression depends on \( h \), checking \((h ? 1 : 2)\) and \((h' ? 1 : 2)\) for equivalence will fail, leading to the program being rejected the program as insecure. The reason for this is that control-flow information about how \( l \) was modified is present in \( \delta \). Concretely, the solver would fail to show validity of the formula \((\text{if } h \text{ then } 1 \text{ else } 2) = (\text{if } h' \text{ then } 1 \text{ else } 2)\), which does not hold, for example, when \( h = \neg h' \). Such information is removed by recursively replacing all occurrences of \((e' ? e_1 : e_2)\) in \( e \) by \((v ? e_1 : e_2)\), where \( v \) is a fresh low variable. A low variable is introduced to force the program to take the same branch in both executions; the variable is fresh so that both branches will be considered by the prover. In fact, the formula \((\text{if } v \text{ then } 1 \text{ else } 2) = (\text{if } v \text{ then } 1 \text{ else } 2)\) is now valid.

The rules in Figure 2.3 force the symbolic execution engine to apply a breadth-first search strategy for analyzing the program. Rule S-OUT considers an output expression \( e \) as secure whenever the state transformation leading to that expression is unaffected by initial tainted values. Indeed, the validity of the first-order formula \( \text{forget}(\delta(e)) = \text{forget}(\tilde{\delta}(e)) \) forces all initial concrete memories that start with the same values for untainted variables (as enforced by the renaming in \( \tilde{\delta}(e) \)) and follow the same execution path (as enforced by function \( \text{forget}(\cdot) \)) to always output the same value to the attacker. This implies that the output observed by the attacker is unaffected by initial tainted values, as required by explicit secrecy. Rule S-IF symbolically evaluates each branch of a conditional and merges the respective symbolic states if the execution of both branches succeeds. Rule S-WHILEUNROLL unrolls a while loop one time. (Unless the number of loop iterations is known a priori, this rule may apply indefinitely, leading to nontermination. Some tools apply loop unrolling for a fixed number of iterations and leave open the possibility of false negatives. This approach is usually taken whenever a tool is used for bug finding, where full automation is more important than possible unsoundness.) Rule S-NEW creates a fresh object location that maps the program variable to that location. Rules S-LOAD loads the symbolic expression contained at some object field by first looking up the object lo-
cation and then the field value of that object location, while rule S-STORE stores the symbolic expression of some program variable to an object field.

Static taint trackers which are more tailored towards verification would attempt either to infer a loop invariant or ask the user to provide one manually. The inference of loop invariants is discussed in the extended version of the paper. We then show soundness of the static enforcement: whenever the symbolic execution engine terminates successfully, the program satisfies explicit secrecy.

2.3.3.1 Definition: A program $c$ satisfies static enforcement, written $\vdash_s c$, iff there exist $\delta', \phi'$ such that $\langle c, \delta_0, \top \rangle \leadsto^* \langle \varepsilon, \delta', \phi' \rangle$.

2.3.3.2 Theorem [Soundness of Static Analysis]: If $\vdash_s c$ then $ES \models c$.

Note that this static enforcement technique can establish security of some programs that are rejected by the dynamic monitoring presented in Section 2.3.1, such as `out (h−h)`. Depending on the power of the SMT solver, static enforcement can fail to validate a program due to dead code. For example, consider `if e then out h else out 0`, where $e$ is a complex expression that always returns 0. If the solver fails to establish that $e$ is equivalent to 0, the program will be rejected. SMT solvers can indeed be expensive, especially for languages pointers and higher-order features. Perhaps this can be remedied by leveraging sound data-flow analysis (à la FlowDroid) or security type systems (as in Section 2.3.2). We have preliminary experiments along these lines with simple programs (no quantifier alternation or non-linear arithmetic); however, over-approximation is needed for richer languages.

Compared to the tainting algorithms used to justify soundness of Andromeda and FlowDroid, the analysis proposed in this section is more precise, i.e., it accepts more secure programs. The rationale behind this choice is that for complex languages as Java, scalability may become an issue as precision increases. Indeed, one central problem in the static analysis of production code is to reconcile precision (object-, field-, context-, flow-sensitivity), scalability (lines of code analyzed) and soundness (FlowDroid is unsound for certain language features). FlowDroid achieves this goal using the IFDS framework by Reps et al. [48] to implement a modular inter-procedural data-flow analysis with on-demand aliasing.
2.4 Related Work

**Taint-tracking Policies.** Our framework is the first to offer a general characterization of policies for taint tracking. The direct precursor of our approach, weak secrecy [62], relies on extracting assignment commands from the original program and, due to its syntactic nature, requires custom-tailored extensions to address such language features as reflection and expressions with side effects.

Chaudhuri et al. [20] implement a calculus for data-flow integrity on Windows Vista. They use a type system and runtime checks to conservatively enforce a tainting policy with respect to an instrumented operational semantics. The instrumented semantics enriches standard semantics with security labels allowing to express explicit secrecy as a safety property. As discussed earlier, explicit secrecy is a hyperproperty [24], hence our condition can be used to justify the soundness of their enforcement mechanism with respect to uninstrumented semantics.

Livshits and Chong [42] address the problem of automatic placement of sanitizers through hybrid taint analyses. They express security policies in terms of sanitizers for a source-sink pair and apply taint tracking to an interprocedural data-flow graph to place the sanitizers. The correctness criteria for tainting algorithms state that sanitizers are placed as described by the policy. Livshits [41] provides a taxonomy of dynamic taint tracking approaches for high-level languages, mainly targeting web security. Taint tracking is an instance of data-flow analysis [39], a method for computing properties about data by observing how it flows through the program. Static and dynamic data-flow analysis, ranging from security types to symbolic executions, have been applied to enforcing security [53, 41, 17]. The semantic notion of explicit secrecy can serve as a criteria for validating the correctness of these approaches.

**Information-flow Policies.** A large array of works on security policies address information-flow control. These policies include the baseline notion of noninterference [34], and variations which account for different policies, languages and computation models [50]. The policies proposed in this paper stand to taint-tracking mechanisms as noninterference-like policies stand to information-flow control mechanisms.

As discussed earlier, there has been work on exploring connec-
tions between information-flow control and taint tracking. Denning and Denning [31] are the first to distinguish between explicit (as in tainting) and implicit flows. Russo et al. [49] show how a lightweight control-flow graph analysis can be combined with explicit flow analysis for non-malicious code. Coppens et al. [28] leverage explicit flow analysis for information-flow security by selective if-conversion to remove branching on secrets in programs by transformation [28]. This technique is a particularly good fit for the implementation of cryptographic algorithms where transforming away branching helps mitigating the timing side channel [13].

A line of work by Vachharajani et al. [60], Graa et al. [35], Beringer [15], and Shen et al. [54] utilizes taint tracking for information-flow control by turning control-flow checks for a source program into data-flow checks of the resulting program. The control-flow checks can be injected by program transformation so that taint tracking on the resulting program can be used for tracking information flow in the source program.

Our knowledge-based condition draws on the notion of gradual release introduced by Askarov and Sabelfeld [11]. Knowledge-based conditions have also been used to provide intuitive semantics for dynamic information-flow policies [8, 61]. This is done by considering attackers that partially forget the observations made during the computation. Although the tainting attacker is forgetful in the sense that it only recalls a single observable event, a precise characterization of tainting policies in terms of forgetful attacker knowledge is non-obvious because it requires to encode attackers that forget the control flow of the program.

Dynamic/Hybrid Enforcement. Taint analysis provides a good balance between implementation effort, bug coverage and performance overhead. This has led to a pervasive application of dynamic taint tracking for enforcing security at all levels of hardware and software stacks. Many systems implement tainting mechanisms in hardware by adding architectural extensions to processors [57, 29, 22, 32] or in software by extending and instrumenting machine code [46, 21, 55, 25], web and mobile applications [33, 45, 58, 42, 30] or high-level languages [4, 3].

Customized hardware and emulators have been used to implement taint tracking policies. Suh et al. [57] introduce a hardware mechanism for dynamically tracking information flows in a pro-
gram. On every instruction, the processor determines whether the result is tainted or not based on the inputs and the instruction type. Various vulnerabilities such as buffer overflows and format strings are captured by disallowing tainted data to be used as instructions or jump target addresses. Crandall et al. [29] introduce Minos, a microarchitecture that implements Biba’s integrity policies at word level. Minos tracks the integrity of all data and it protects from control flow hijacking by checking taintedness whenever a program uses that data to transfer control. TaintBoch [22] uses tainting to track sensitive data across operating system, language, and application boundaries, thus permitting analysis at a whole system level. In principle, all these approaches can handle self-modifying code and thus benefit from our security condition to justify their correctness.

Many tools use dynamic instrumentation of machine code to monitor system activities through taint tracking. TaintTrace [21] uses the DynamoRIO framework [1] to instrument machine code. Similarly, TaintCheck [46] offers both Valgrind-based [5] and DynamoRIO-based implementations for the same purpose. Notably, the CMU Binary Analysis Platform [18] and the BitBlaze toolchain [55] offer a unified platform for static and dynamic security analysis of binary code. They explicitly represent all side effects of machine instructions in an intermediate language (IL), for which various forms of data-flow analysis, including taint tracking, are implemented. Assuming correctness of the transformation between binary code and IL code, one may use the well-defined semantics of IL to show soundness for the data-flow analysis implemented by these tools.

Dynamic taint tracking has been successfully used for bug finding in mobile and web applications. TaintDroid [33] is an extension to the Android platform that monitors how a potentially untrusted application uses user data. TaintDroid uses dynamic tainting to automatically track propagation of sensitive data through program variables or files and ensures that tainted data are transmitted over the network only with user’s consent.

Perl’s taint mode [4] is one of the first applications of dynamic taint tracking to high-level languages. When the interpreter is run in this mode, several data sources such as environment variables or command-line parameters produce tainted values. Tainted data may not be used by any command that invokes a sub-shell, nor
in any command that modifies files, directories, or processes, with a few exceptions: (1) arguments to `print` and `syswrite` are not checked for taintedness; (2) symbolic methods and symbolic sub references are not checked for taintedness; and (3) hash keys are never tainted. These exceptions allow for laundering (i.e., declassifying) tainted values either by using them as hash keys or by regexp-matching against them and using the sub-match strings $1, $2, etc.

Several generic frameworks offer customizable data-flow analysis and policies to account for the lack of generality and the high performance overhead in traditional taint-tracking systems. GIFT [40] is a compiler for programs written in C that takes programmer-specified rules for taint initialization, propagation and combination, and automatically instruments programs so as to execute these rules as part of the program execution. Chang et al. [19] build a compiler to transform untrusted programs into policy-enforcing programs. The compiler can be reconfigured to support new analyses and policies and it uses static analysis reduce the amount of data that must be dynamically tracked. Our work lays ground for turning informal soundness claims in this work into formal ones.

**Static Enforcement.** Purely static analysis has been proposed to tracking explicit flows in various application domains. Bodden et al. [7] present FlowDroid, a context-, flow-, field-, object-sensitive static taint analysis tool for Android applications. FlowDroid analysis is claimed to be conservative and sound with respect to the analysis it implements. However, unsoundness can arise in case the sequential consistency of thread execution is broken, or through native methods that are possibly modeled incorrectly.

Sridharan et al. [56] present F4F, a system for effective taint analysis of framework-based web applications. F4F initially analyses application code and configuration files to generate specifications of framework-related behaviors, and then uses a taint engine to perform more precise analysis of the framework-based applications.

Pixy [38] is a static taint analysis tool for detecting cross-site scripting and SQL injection vulnerabilities in PHP programs. It implements flow-sensitive, interprocedural and context-sensitive data-flow analysis to discover vulnerable program points.
The Parfait bug checker [52] builds on top of the LLVM compiler, performing a staged dependency analysis which takes into account both data and control dependencies. The tool is inherently tunable to different precision levels and it allows for demand driven analysis.

Balliu et al. [12] implement an automated tool for information-flow analysis of machine code. The tool transforms ARMv7 binaries into an architecture-independent format using the BAP platform [18] by means of a verified translator. This approach leverages symbolic execution and SMT solvers for machine-code verification, hence it is possible to extend the symbolic algorithm in Section 2.3.3 to that setting and thus verify explicit secrecy.

2.5 Conclusion

We have presented a generic semantic framework for reasoning about explicit flows as tracked by taint checking. The framework generalizes previous work by giving a condition that enables reasoning about what taint tracking guarantees and covers a wide range of settings, declassification/sanitization policies, and including low-level machines and languages with reflection. The framework has allowed us to formally compare security conditions, establishing a relation to such known characterizations as noninterference and gradual information release. We have demonstrated the usefulness of the framework by instantiations to high- and low-level languages.

Explicit secrecy contributes to understanding of what is achieved by popular taint tracking tools, both dynamic and static. We have demonstrated that taint mechanisms at the core of dynamic tools such as BAP [18] and BitBlaze [55] and languages as Perl [4] and Ruby [3] are sound with respect to explicit secrecy. Similarly, we have showed the soundness of a mechanism reminiscent of FlowDroid [7] to guarantee the absence of explicit flows.

Our work opens up promising opportunities to formalizing informal soundness claims made in the taint tracking literature, following the initial steps in Section 2.3. An important track for future work is generalizing the soundness results to include declassification/sanitization, as defined in Section 2.2.3. Section 2.4 discusses several future tracks for applying our approach to the static and dynamic enforcement mechanisms from the literature. The expected
benefits are greater confidence in these mechanisms and possibly discovering corner cases where soundness can be improved.

Acknowledgments. This work was funded by the European Community under the ProSecuToR project and the Swedish research agencies SSF and VR.

Bibliography


[53] 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.


2. A Detailed Semantics

2. A.1 Language Semantics

The semantics of while programs is given by the relation $\rightarrow$ defined below:

\[
\begin{align*}
(\text{skip}, m) & \rightarrow (\varepsilon, m) & \text{(E-Skip)} \\
(x := e, m) & \xrightarrow{\text{assign}} (\varepsilon, m[x \mapsto n]) & \text{(E-Assign)} \\
(out e, m) & \xrightarrow{n} (\varepsilon, m) & \text{(E-Out)} \\
\text{if } e \text{ then } c_1 \text{ else } c_2, m & \rightarrow (c_1, m) & \text{(E-IfTrue)} \\
\text{if } e \text{ then } c_1 \text{ else } c_2, m & \rightarrow (c_2, m) & \text{(E-IfFalse)}
\end{align*}
\]
2.A. Detailed Semantics

\[
m(e) = n \quad n \neq 0 \quad (\text{while } e \text{ do } c, m) \rightarrow (c; \text{while } e \text{ do } c, m) \quad (E\text{-WhileTrue})
\]

\[
m(e) = n \quad n = 0 \quad (\text{while } e \text{ do } c, m) \rightarrow (\varepsilon, m) \quad (E\text{-WhileFalse})
\]

\[
(c_1, m) \xrightarrow{\alpha}{\beta} (c'_1, m')
\]

\[
(c_1 ; c_2, m) \xrightarrow{\alpha}{\beta} (c'_1 ; c_2, m') \quad (E\text{-Seq})
\]

\[
(\varepsilon ; c_2, m) \xrightarrow{\beta} (c_2, m) \quad (E\text{-SeqEmpty})
\]

Subscripts on \(\rightarrow\) are used to define weak secrecy, but do not influence the execution of a program in any other way.

2.A.2 Extended Language Semantics

The semantics of Java-like programs is given by the extended relation \(\rightarrow\) defined below:

\[
(c, m) \xrightarrow{\alpha}{\beta} (c', m')
\]

\[
(c, m, h) \xrightarrow{\alpha}{\beta} (c', m', h) \quad (E\text{-Rule})
\]

\[
l \in \text{Loc fresh} \quad m' = m[x \mapsto l] \quad h' = h[l \mapsto \text{null}] \quad (x := \text{new Object}(), m, h) \xrightarrow{\text{new Object}} (\varepsilon, m', h') \quad (E\text{-New})
\]

\[
l = m(y) \quad m' = m[x \mapsto h(l, f)] \quad (x := y.f, m, h) \xrightarrow{y.f} (\varepsilon, m', h) \quad (E\text{-Load})
\]

\[
l = m(x) \quad h' = h[h(l, f) \mapsto y] \quad (x.f := y, m, h) \rightarrow (\varepsilon, m, h') \quad (E\text{-Store})
\]

2.A.3 Semantics for Control-Flow Gradual Release

The semantics for control-flow gradual release use the following modified rules for branching:

\[
m(e) = n \quad n \neq 0 \quad (\text{if } e \text{ then } c_1 \text{ else } c_2, m) \xrightarrow{r(n)} (c_1, m) \quad (E\text{-IfTrue})
\]
Chapter 2: Explicit Secrecy

\[
\begin{align*}
    & m(e) = n \quad n = 0 \quad \text{(E-IFFALSE)} \\
    & \text{(if } e \text{ then } c_1 \text{ else } c_2, m) \xrightarrow{r(n)} (c_2, m) \\
    & m(e) = n \quad n \neq 0 \quad \text{(E-WHILETRUE)} \\
    & \text{(while } e \text{ do } c, m) \xrightarrow{r(n)} (c; \text{while } e \text{ do } c, m) \\
    & m(e) = n \quad n = 0 \quad \text{(E-WHILEFALSE)} \\
    & \text{(while } e \text{ do } c, m) \xrightarrow{r(n)} (\varepsilon, m)
\end{align*}
\]

\(r(n)\) denotes a release event that where the value \(n\) is released.

2.B Machine Code

2.B.1 Syntax and Semantics

\[
\langle c \rangle ::= \text{nop} | \text{const } i \text{ } r | \text{mov } r_{\text{src}} \text{ } r_{\text{dst}} | \text{op} r_1 \text{ } r_2 \text{ } r_{\text{dst}} | \text{load } r_{\text{addr}} \text{ } r_{\text{dst}} | \text{store } r_{\text{addr}} \text{ } r_{\text{val}} | \text{jump } r | \text{jal } r | \text{bnz } r \text{ } i | \text{halt}
\]

2.B.1.1 Definition: A machine configuration is a triple consisting of memory state \(m = \text{mem} : \text{Addr} \rightarrow \mathbb{Z}\), a register state \(r = \text{reg} : \text{Reg} \rightarrow \mathbb{Z}\) and a program counter \(p = \text{pc}\). We denote such a configuration by \((m, r, p)\).

Figure 2.B.1 shows the small-step semantics of the language.

2.C Proofs

2.C.1 Specifying Explicit Flows

Proof of Lemma 2.2.2.2: We prove the statement by induction on \(\text{cfg} \xrightarrow{\alpha} \text{cfg}'\): Reflexive case: In this case it holds that \(\text{cfg} = \text{cfg}'\), \(\alpha = \varepsilon\), \(f(s) = (s, \varepsilon)\). The statement then follows straightforwardly:

\[
f(\text{state} (\text{cfg})) = (\text{state} (\text{cfg}), \varepsilon) = (\text{state} (\text{cfg}'), \alpha)
\]
2.C. Proofs

Transitive case: Assume \( \text{cfg} \xrightarrow{\alpha} \text{cfg}'' \xrightarrow{\beta} \text{cfg}' \). By induction hypothesis we have \( f(\text{state}(\text{cfg})) = (\text{state}(\text{cfg}''), \alpha) \). We need to show that \( (g \circ f)(\text{state}(\text{cfg})) = (\text{state}(\text{cfg}'), \alpha.\beta) \), where \( \alpha.\beta \) denotes the concatenation of \( \alpha \) and \( \beta \). By definition of extracted functions:

\[
g(\text{state}(\text{cfg}')) = (s', \beta_1)
\]

where \( \langle \text{com}(\text{cfg}''), \text{state}(\text{cfg}'') \rangle \xrightarrow{\beta} \text{cfg}'_1 \) and \( s' = \text{state}(\text{cfg}'_1) \). By the properties of \( \langle \cdot, \cdot \rangle \), \( \text{state}(\cdot) \), and \( \text{com}(\cdot) \) has, it holds that \( \langle \text{com}(\text{cfg}''), \text{com}(\text{cfg}'') \rangle = \text{cfg}'' \). Together with the assumption of determinism, this yield that \( \beta = \beta_1 \) and \( \text{cfg}'_1 = \text{cfg}' \) and hence the statement follows, using the definition of \( \circ \).

\[ \square \]

2.C.1.1 Lemma: Whenever \((c, m_0) \xrightarrow{\text{ao}} (c', m'_0) \land (d, m) \xrightarrow{\text{g}} (\varepsilon, m')\), it holds that \( g = f \), where \( d \) is the extracted program and \( f \) is the state transformer.

Proof: By induction on \( \xrightarrow{\text{ao}} \): 

Reflexive case: In this we have \( d = \varepsilon \) and \( f(m) = (m, \varepsilon) \). Similarly, \( \varepsilon \) can only evaluate (in 0 steps) to itself, producing an empty trace and \( g(m) = (m, \varepsilon) \).

Transitive case: Assume \((c, m_0) \xrightarrow{\text{ao}} (c', m'_0) \xrightarrow{\alpha_0} (c'', m''_0)\). By induction hypothesis, we have that whenever \((d, m) \xrightarrow{\text{g}} (\varepsilon, m')\), then \( g = f \).

If \( f' = (\lambda m. (m, \varepsilon)) \), then \( d' = \varepsilon \) and the statement follows trivially from the induction hypothesis. In the other cases, the first command in \( c' \) is either an output statement or an assignment.

If \( c' = x := e ; \tilde{c}' \) or \( c' = x := e \) we have \( d' = x := e \) and \( f'(m) = (m[x \mapsto m(e)], \varepsilon) \). In this case, we have that \((d ; x := e, m) \xrightarrow{\text{g}} (x := e, m') \xrightarrow{\alpha'} (\varepsilon, m'') \) with \( g'(m) = (m[x \mapsto m(e)], \varepsilon) = f'(m) \), as desired.

If \( c' = \text{poute} ; \tilde{c}' \) or \( c' = \text{out} e \), then \( d' = \text{out} e \) and \( f'(m) = (m, [m(e)]) \). Similarly, we have that \((d ; \text{out} e, m) \xrightarrow{\text{g}} (\text{out} e, m') \xrightarrow{\alpha'} (\varepsilon, m') \) with \( g'(m) = (m, [m(e)]) \) as desired.

\[ \square \]

2.C.1.2 Lemma: Whenever \((c, m_0) \xrightarrow{\text{ao}} (c', m'_0) \land f(m) = (m', \alpha)\), we have \((d, m) \xrightarrow{\alpha} (\varepsilon, m')\).
**Proof:** We proceed by induction on $\frac{\alpha_0}{f,d}^*$.

Reflexive case: In this case $f(m) = (m, \varepsilon)$ and $d = \varepsilon$. In that case $(\varepsilon, m) \not\xrightarrow{\varepsilon}^* (\varepsilon, m)$ trivially holds.

Transitive case: Assume $(c, m_0) \xrightarrow{\alpha_0}_{f,d}^* (c', m_0')$ and $(c', m_0') \xrightarrow{\alpha'}_{f',d'}^* (c'', m_0'').$ Assume that $f(m) = (m', \alpha)$ and $f'(m') = (m'', \alpha')$. By induction hypothesis, it holds that $(d, m) \xrightarrow{\alpha_0}^* (\varepsilon, m')$. By the semantics of evaluation, we get $(d ; d', m) \xrightarrow{\alpha}^* (d', m') \xrightarrow{\alpha''}^* (\varepsilon, m'')$. By definition of $f'$ we have that $(\varepsilon', m') \xrightarrow{\alpha'}^* (c'', m'')$ for some $c''$. If the first command in $c'$ is neither an output nor an assignment, no trace and no command is produced and the conclusion follows.

For outputs and assignments, note that the next evaluation step only depends on the first command in $c'$, which is the command that is extracted as $d'$. In this case, we get $\alpha'' = \alpha'$ and $m' = m''$ by determinism. \[\square\]

**Proof of Theorem 2.2.4.1:** ($\Rightarrow$): Assume $ES \models c, (c, m_0) \xrightarrow{\alpha_0}_{f,d}^* (c_0, m_0')$. We have to show that $NI \models d$. We proceed by induction $(c, m_0) \xrightarrow{\alpha_0}_{f,d}^* (c_0, m_0')$:

Reflexive case: In this case we have $d = \varepsilon$ and noninterference for $\varepsilon$ holds trivially.

Transitive case: Assume $(c, m_0) \xrightarrow{\alpha_0}_{f,d}^* (c_0', m_0') \xrightarrow{\alpha'}_{f',d'}^* (c_0, m_0')$. By induction hypothesis, $NI \models d$. We have to show noninterference for $d ; d'$. Let $(d ; d', m_1) \xrightarrow{\alpha_1, \alpha'_1}_{f'_1 \circ f_1}^* (\varepsilon, m_1')$ and $(d ; d', m_2) \xrightarrow{\alpha_2, \alpha'_2}_{f'_2 \circ f_2}^* (\varepsilon, m_2')$ with $m_1 =L m_2$. From Lemma 2.2.1.1 we obtain that $f'_1 \circ f_1 = f' \circ f$ and $f'_2 \circ f_2 = f' \circ f$. From $ES \models c$, we obtain that $\forall m.[m]_L \subseteq k_e(m, f' \circ f)$. In particular, we also have that $m_2 \in k_e(m_1, f' \circ f)$, i.e. $\pi_2((f' \circ f)(m_2)) = \pi_2((f' \circ f)(m_1))$. Together with Lemma 2.2.2.2, this yields that $\pi_2((f' \circ f)(m_1)) = \alpha_1 = \alpha_2 = \pi_2((f' \circ f)(m_2))$, as desired.

($\Leftarrow$): Assume $WS \models c$ and $(c, m_0) \xrightarrow{\alpha_0}_{f_0}^* (c_0', m_0')$. We need to show that $\forall m_1, [m_1]_L \subseteq k_e(m_1, f)$. Assume $m_2 \in [m_1]_L$. We proceed by induction on $(\frac{\alpha_0}{f_0}^*)$:

Reflexive case: In this case, we have $f(m) = (m, \varepsilon)$. I.e. $k_e(m_1, f) = [m_1]_L$ and the statements holds trivially.
Transitive case: Assume \((c, m_0) \xrightarrow{\alpha_0 \circ \ell_0} (c', m_0')\). By induction hypothesis it holds that \(m_2 \in k_e(m_1, f)\). We need to show that \(m_2 \in k_e(m_1, g \circ f)\). Let \(f(m_1) = (m''_0, \alpha_1)\), \(f(m_2) = (m''_2, \alpha_2)\), \(g(m''_0) = (m'_1, \alpha'_1)\), and \(g(m''_2) = (m'_2, \alpha'_2)\). Due to the induction hypothesis, we get that \(\alpha_1 = \alpha_2\).

Using Lemma 2.C.1.2, we get \((d; d', m_i) \xrightarrow{\alpha_i, \alpha'_i} (\varepsilon, m'_i)\) for \((i = 1, 2)\). Since \(WS \models c\), we have \(NI \models d; d'\) and hence \(\alpha_1, \alpha'_1 = \alpha_2, \alpha'_2\). Together with \(\alpha_1 = \alpha_2\), this entails \(\alpha'_1 = \alpha'_2\) and hence \(m_2 \in k_e(m_1, f)\) as desired.

Control-Flow Gradual Release: Traces for the following proofs include release events as generated by control-flow gradual release.

\subsection*{2.C.1.3 Lemma} If \((c, m_0) \xrightarrow{\alpha_0 \circ \ell_0} (c', m_0')\) and \((c, m) \xrightarrow{\alpha_0} (c', m')\), then \(\exists \tilde{m}', f'.(c', m') \xrightarrow{\varepsilon} (c'_0, \tilde{m}') \land f_0 = f' \circ f\).

\textbf{Proof}: Straight-forward induction on evaluation relation.

\textbf{Proof of Theorem 2.2.5.3}: Assume \(ES \models c\) and \((c, m_0) \xrightarrow{\ell_0 \circ \ell_0} (c'_0, m'_0)\) and \(\forall n. \alpha_0 \neq \text{rel}(n)\). Hence, \(c' = \text{out } e\) or \(c' = \text{out } e; \overline{c'}\). Moreover, assume \(m \in k(c, m_0, \ell_0)\), i.e. \((c, m) \xrightarrow{\ell_0} (c', m')\).

From Lemma 2.C.1.3, we get \((c', m') \xrightarrow{\varepsilon} (c'_0, \tilde{m}') \xrightarrow{\alpha} (c''_0, m'')\) for some \(m''\) such that \(f_0 = f' \circ f\) and \(g = g_0\). Since \(ES \models c\), we have \(m \in k_e(m_0, g \circ f_0)\), i.e. \((g \circ f_0)(m_0) = \pi_2(g \circ f_0)(m)\) (we write \(a = f b\) for \(f(a) = f(b)\)).

Together with Lemma 2.2.2.2, this yields \(f(m) = (m', \ell_0), f_0(m_0) = (m''_0, \ell_0), g_0(m'_1) = (m'', \alpha), \) and \(g_0(m'_2) = (m''_2, \alpha_0)\). Since \((g_0 \circ f_0)(m_0) = \pi_2(g_0 \circ f_0)(m)\), we have \(\ell_0, \alpha = \ell_0, \alpha\) and hence \(\alpha = \alpha_0\). This entails \(m \in k(c, m_0, \ell_0, \alpha_0)\) as desired.

\subsection*{2.C.2 Enforcement}
2.C.2.1 Lemma [Faithfulness of tainting]: Given a program \( c \), an initial state \( s \) and a labeling function \( \tau \), if \((c, s, \tau) \xrightarrow{\alpha} (c', s', \tau')\), then \((c, s) \xrightarrow{\alpha} (c', s')\).

**Proof:** By rule inspection. The rules for dynamic taint tracking are a restricted version of small-step operational semantics rules. □

In what follows, we write \((c, s, \tau) \xrightarrow{\alpha} (c', s', \tau')\) whenever \((c, s, \tau) \xrightarrow{\alpha} (c', s', \tau')\) and \((c, s) \xrightarrow{\alpha} (c', s')\).

2.C.2.2 Lemma: Let \((c, m, \tau) \xrightarrow{\alpha} (c', m', \tau')\) then for all \( m^* \), \( m^* \mathrel{\sim} m \), we have \( \pi_i(f(m)) = \tau \mathrel{\sim} \pi_i(f(m^*)) \) for \( i \in \{1, 2\} \).

**Proof:** We proceed by induction on \( \xrightarrow{\alpha} \):

Reflexive case: We have \( f(m) = (m, \varepsilon) \) and \( \tau = \tau' \), hence the statements holds trivially.

Transitive case: Assume \((c, m, \tau) \xrightarrow{\alpha} (c'', m'', \tau'')\) and \((c'', m'', \tau'') \xrightarrow{\alpha} (c', m', \tau')\).

By induction hypothesis it holds that for all \( m^* \), \( m^* \mathrel{\sim} m \), we have \( \pi_i(h(m)) = \tau \mathrel{\sim} \pi_i(h(m^*)) \) for \( i \in \{1, 2\} \). We first show that \( \pi_1(g \circ h(m)) = \tau \mathrel{\sim} \pi_1(g \circ h(m^*)) \) by case analysis on \( g \) and rules in Figure 2.1. The only interesting case is rule T-Assign. W.l.o.g. assume the current statement is \( x := e \), then \( m' = g(m'') = (m''[x \mapsto m''(e)], \varepsilon) \) and \( \tau' = \tau''[x \mapsto \tau''(e)] \). Let \( m'' = \pi_1(h(m^*)) \), then \( m'' = \pi_1(g(m^*)) \), we show that \( m^* = \tau \mathrel{\sim} m' \).

There are 2 cases to consider: (1) if \( \tau''(e) = H \) then \( \tau'(x) = H \), hence \( m'' = \tau \mathrel{\sim} m' \). (2) otherwise \( \tau''(e) = L \), thus forall \( y \in \text{vars}(e) \), \( \tau''(y) = L \), which implies that \( m'(x) = m''(x) \), i.e., \( m' = \tau \mathrel{\sim} m'' \).

We now show that \( \pi_2(g \circ h(m)) = \tau \mathrel{\sim} \pi_2(g \circ h(m^*)) \) by case analysis on \( g \) and rules in Figure 2.1. The only interesting case is rule T-OUTL. Assume the current statement is \( \text{out} e \) and \( \tau''(e) = L \). Since \( m'' = \tau \mathrel{\sim} m'' \), then \( m'' = m''(e) \), hence \( \pi_2(g \circ h(m)) = \tau \mathrel{\sim} \pi_2(g \circ h(m^*)) \).

**Proof of Theorem 2.3.1.1:** Let \((c, m, \tau) \not \xrightarrow{\alpha} \emptyset \) and \((c, m, \tau) \xrightarrow{\alpha} (c', m', \tau')\). We need to show that \( \forall m_1, [m_1]_\tau \subseteq k_e(m_1, f) \), which follows by Lemma 2.C.2.2. □
2.C.2.3 Lemma: Let
\[ (\text{mem}[pc \mapsto is], \text{reg}, pc, \tau) \overset{\alpha^*}{\rightarrow} (\text{mem}', \text{reg}', pc', \tau') \]
then for all \( s^* = (m^*, \text{reg}^*) \), \( s^* =_\tau s \) and \( s^* \subseteq S_0(c) \), we have \( \pi_i(f(s)) =_\tau \pi_i(f(s^*)) \) for \( i \in \{1, 2\} \).

Proof: We proceed by induction on \( \overset{\alpha^*}{\rightarrow} f \):

Reflexive case: We have \( f(s) = (s, \varepsilon) \) and \( \tau = \tau' \), hence the statements holds trivially.

Transitive case: Assume
\[ (\text{mem}[pc \mapsto is], \text{reg}, pc, \tau) \overset{\alpha^*}{\rightarrow} (\text{mem}'', \text{reg}'', pc'', \tau'') \]
\[ \overset{\alpha'}{\rightarrow} (\text{mem}', \text{reg}', pc', \tau') \]

By induction hypothesis it holds that for all \( s^* \), \( s^* =_\tau s \) and \( s^* \subseteq S_0(c) \), we have \( \pi_i(h(s)) =_\tau'' \pi_i(h(s^*)) \) for \( i \in \{1, 2\} \). We first show that \( \pi_1(g \circ h(s)) =_\tau \pi_1(g \circ h(s^*)) \) by case analysis on \( g \) and rules in Figure 2.2.

Case T-STORE: Let \( \text{com}(cfg'') = (\text{store} \ r_a \ r_v, pc'') \), then \( s' = g(s'') = ((\text{mem}'', reg''[r_a] \mapsto reg''[r_v]), \varepsilon) \), \( a = reg''[r_a] \) and \( \tau' = \tau''[a \mapsto \tau''(r_v) \sqcup \tau''(a)] \). Let \( s''^* = \pi_1(h(s'')) \), then \( s''^* =_\tau'' s'' \) by the induction hypothesis. Let also \( s''^* = \pi_1(g(s''^*)) \), we show that \( s''^* =_\tau s' \). There are 2 cases to consider: (1) if \( \tau''(r_v) = \text{H} \) or \( \tau''(a) = \text{H} \), then \( \tau'(a) = \text{H} \), hence \( s''^* =_\tau s' \). (2) otherwise \( \tau''(r_v) = \tau''(a) = \text{L} \), hence \( \text{mem}'(a) = \text{mem}''(a) \), i.e., \( \text{mem}' =_\tau \text{mem}'' \), since \( \text{reg}''(r_v) = \text{reg}''(r_v) \). Therefore \( s''^* =_\tau s''^* \).

Case T-LOAD: Let \( \text{com}(cfg'') = (\text{load} \ r_a \ r_d, pc'') \), then \( s' = g(s'') = ((\text{mem}'', \text{reg}''[r_d \mapsto v]), \varepsilon) \), \( v = \text{mem}''[\text{reg}''[r_a]] \) and \( \tau' = \tau''[r_d \mapsto \tau''(r_a) \sqcup \tau''(\text{reg}''[r_a])] \). Let \( s''^* = \pi_1(h(s'')) \), then \( s''^* =_\tau'' s'' \) by the induction hypothesis. Let also \( s''^* = \pi_1(g(s''^*)) \), we show that \( s''^* =_\tau s' \). There are 2 cases to consider: (1) if \( \tau''(r_a) = \text{H} \) or \( \tau''(\text{reg}[r_a]) = \text{H} \), then \( \tau'(r_d) = \text{H} \), hence \( s''^* =_\tau s' \). (2) otherwise \( \tau''(r_a) = \tau''(\text{reg}[r_a]) = \text{L} \) and \( \text{mem}'(v) = \text{mem}''(v) \), hence \( \text{reg}''(r_d) = \text{reg}''(r_d) \). Therefore \( s''^* =_\tau s''^* \).

Case T-JAL: Let \( \text{com}(cfg'') = (\text{jal} \ r, pc'') \), then \( s' = g(s'') = ((\text{mem}'', \text{reg}''[sp] \mapsto pc''), \varepsilon) \), \( a = \text{reg}''[sp] \) and \( \tau' = \tau''[a \mapsto \tau''(sp) \sqcup \tau''(pc'')] \). Let \( s''^* = \pi_1(h(s'')) \), then \( s''^* =_\tau'' s'' \) by the induction hypothesis. Let also \( s''^* = \pi_1(g(s''^*)) \), we show that \( s''^* =_\tau s' \). There are 2 cases to consider: (1) if \( \tau''(sp) = \text{H} \) or
\( \tau''(pc') = H \), then \( \tau'(a) = H \), hence \( s'^* =_{\tau'} s' \). (2) otherwise \( \tau''(sp) = \tau''(pc') = L \), hence \( mem'(a) = mem'^*(a) \), i.e., \( mem' =_{\tau'} mem'^* \), since \( reg''(sp) = reg''*(sp) \) and \( reg''(pc') = reg''*(pc') \). Therefore \( s' =_{\tau'} s'^* \).

Other cases are similar.

We now show that \( \pi_2(g \odot f(m)) =_{\tau'} \pi_2(g \odot f(m^*)) \) by case analysis on \( g \) and rules in Figure 2.2. The only interesting case is rule T-OUTL. Assume \( com(cfg'') = (\text{out } r, pc'') \) and \( \tau''(r) = L \). Then \( reg''(r) = reg''*(r) \) and \( \tau' = \tau'' \), hence \( \pi_2(g \odot h(s)) =_{\tau'} \pi_2(g \odot h(s^*)) \).

\[ \Box \]

**Proof of Theorem 2.3.2.1:** Recall that \( c = (is, pc) \), \( s = (mem, reg) \). Suppose \( (mem[pc \mapsto is], reg, pc, \tau) \not\xrightarrow{\tau'} f \) and \( (mem[pc \mapsto is], reg, pc, \tau) \xrightarrow{\alpha} (mem', reg', pc', \tau') \). We show that \( \forall s_1, [s_1]_\tau \cap S_0(c) \subseteq k_e(s_1, f) \), where the set of valid initial states is \( S_0(c) = \{ s \mid \exists s'.state((c, s')) = s \} \). The theorem follows immediately by Lemma 2.C.2.3.

\[ \Box \]

**Static Taint Tracking.** We prove soundness of the static enforcement presented in Section 2.3.3 by first relating the symbolic execution rules given in that section to a more conventional small-step symbolic execution semantics:
If we set $State = Sym = (Var \to Exp) \cup (Loc \times Fld \to Exp)$, i.e. the set of symbolic states, the explicit secrecy framework can be used to extract functions $f : Sym \to Sym \times Exp^*$ for symbolic evaluation steps. We first connect such functions to ordinary state transformers. Such a function can be turned into a state transformer $f_0 : Mem \to (Mem \times \mathbb{Z}^*)$:

2.C.2.4 Definition: For $f : Sym \to Sym \times Exp^*$, we define $F(f)$:

\[ F(f) : \]
Mem → Mem × Z* by
\[ F(f)(m) = (m \circ (\pi_1(f(\delta_0))), m(\pi_2(f(\delta_0)))) \]

We write \( m([e_1, \ldots, e_n]) \) to denote \([m(e_1), \ldots, m(e_2)]\).

We connect the modified symbolic execution semantics to static enforcement and normal runs by the following lemmas:

2.C.2.5 Lemma: If \( \vdash s c \), then \( \langle c, \delta_0, \varphi_0 \rangle \not\rightarrow^* f \).

Proof: Straightforward by noting that they only differ on rules for if statements. After merging the symbolic states, both executions are considered by using \( \text{forget}(\cdot) \) when checking output statements.

\( \square \)

2.C.2.6 Lemma: If \( (c, m) \xrightarrow{f_0} (c, m') \) and \( \vdash s c \), then
\[ \exists \delta', \varphi'. \langle c, \delta_0, \varphi_0 \rangle \rightarrow^* \langle c', \delta', \varphi' \rangle \land F(f) = f_0 \land m \vdash \varphi \]

where \( m \vdash \varphi \) denotes that formula \( \varphi \) is true in memory \( m \); i.e. \( m(\varphi) \neq 0 \).

Proof: Assume \( \vdash s c \) and \( (c, m) \xrightarrow{f_0} (c, m') \). We proceed by induction on the latter.

Reflexive case: In this case, it holds that \( f_0(m) = (m, \varepsilon) \) and similarly, \( f(\delta) = (\delta, \varepsilon) \) and the conclusion follows immediately.

Transitive case: Assume \( (c, m) \xrightarrow{f_0} (c', m'_0) \xrightarrow{g_0} (c'', m''_0) \). By induction hypothesis, we obtain \( \delta', \varphi' \), and \( f \) such that
\[ \langle c, \delta_0, \varphi_0 \rangle \rightarrow^* \langle c', \delta', \varphi' \rangle \land F(f) = f_0 \land m_0 \vdash \varphi' \]. Note that this entails that \( \forall x. m(\delta'(x)) = m'(x) \).

We show \( \exists \delta'', \varphi'', g. \langle c', \delta', \varphi' \rangle \rightarrow^* \langle c'', \delta'', \varphi'' \rangle \land F(g) = g_0 \land m_0 \vdash \varphi'' \) by induction on \( \rightarrow \). The conclusion then follows together with the induction hypothesis.

Case E-Skip: It holds that \( g_0(m) = (m, \varepsilon) \). We apply A-Skip to match the execution step; the extracted function is \( g(\delta) = (\delta, \varepsilon) \) and one checks that \( F(g) = g_0 \).

Case E-Assign: \( c' = x := e, c'' = \varepsilon, g_0(m) = (m[x \mapsto m(e)], \varepsilon) \). We match the step by applying A-Assign and obtain \( \varphi'' = \varphi', \delta'' = \delta'[x \mapsto \delta'(e)], g(\delta) = (\delta[x \mapsto \delta(e)], \varepsilon) \). For \( F(g) = g_0 \),
we note that equality on the second components follows directly. For the first components we note:

\[ \pi_1(\mathcal{F}(g)(m)) = m \circ \pi_1(g(\delta_0)) = m \circ \delta_0[x \mapsto e] \]

We proceed by case distinction on the argument. If \( y \neq x \), then:

\[ \pi_1(\mathcal{F}(g)(m))(y) = m(\delta_0[x \mapsto e](y)) = m(\delta_0(y)) = m(y) = m[x \mapsto e](y) = \pi_1(g_0(m))(y) \]

If \( y = x \):

\[ \pi_1(\mathcal{F}(g)(m))(x) = m(\delta_0[x \mapsto e](x)) = m(e) = m[x \mapsto e](x) = \pi_1(g_0(m))(x) \]

The cases for E-New, E-Load, and E-Store follow similarly.

Case E-Out: It holds that \( c' = \textbf{out} \; e, \; g_0(m) = (m, [m(e)]) \), \( c'' = \varepsilon \). With Lemma 2.C.2.5, we obtain that \( \langle c, \delta_0, \varphi_0 \rangle \rightarrow^* \langle x := e, \delta', \varphi' \rangle \not\rightarrow 4 \). This entails that we cannot apply rule A-OutFail and hence \( SMT \vdash \delta' e = \delta'(e) \). We then apply A-Out to obtain \( \langle x := e, \delta', \varphi' \rangle \rightarrow \langle \varepsilon, \delta', \varphi' \rangle \) with \( g(\delta) = (\delta, [\delta(e)]) \). Clearly, \( g_0 \) and \( \mathcal{F}(g) \) coincide on their first components. For the second components we compute:

\[ \pi_2(\mathcal{F}(g)(m)) = m([\delta_0(e)]) = m([e]) = [m(e)] = \pi_2(g_0(m)) \]

Case E-IfTrue: In this case, we have \( c' = \textbf{if} \; e \; \textbf{then} \; c_1 \; \textbf{else} \; c_2, \; g_0(m) = (m, \varepsilon), \; c'' = c_1, \; m'(e) \neq 0 \). Hence \( m(\delta'(e)) \neq 0 \), and together with soundness of the SMT solver, it holds that SMT \( \not\vdash \neg\delta'(e) \). Using this, we apply A-IfTrue to obtain \( \langle c', \delta', \varphi' \rangle \rightarrow^* \langle c_1, \delta', \varphi'' \rangle \) with \( \varphi'' = \varphi' \land \delta'(e) \), and \( g(\delta) = (\delta, \varepsilon) \). \( g_0 = \mathcal{F}(g) \) follows directly. \( m \vdash \varphi'' \) follows from \( m(\delta'(e)) \neq 0 \).

The cases for E-IFFalse, E-WhileTrue, and E-WhileFalse follow similarly.

The rules for sequential composition follow straightforwardly from the induction hypothesis.
Moreover, we obtain soundness with respect to the function extracted in the modified semantics:

**2.C.2.7 Lemma:** If \( \langle c, \delta_0, \varphi_0 \rangle \xrightarrow{f}^* \langle c', \delta', \varphi' \rangle \), then \( \forall m. [m]_L \subseteq k_e(m, F(f)) \).

**Proof:** By induction on \( \xrightarrow{f}^* \). The reflexive case follows easily.

For the transitive case assume \( \langle c, \delta_0, \varphi_0 \rangle \xrightarrow{f}^* \langle c', \delta', \varphi' \rangle \xrightarrow{g} \langle c'', \delta'', \varphi'' \rangle \). By induction hypothesis we get \( \forall m. [m]_L \subseteq k_e(m, F(f)) \). We need to show that \( \forall m. [m]_L \in k_e(m, g(F(g)) \cap F(f)) \). Assume \( m_1 \in [m]_L \). The only interesting case for step \( \xrightarrow{g} \) is using the rule A-OUT.

In this case, it holds that \( g(\delta) = (\delta, [\delta(e)]) \) and hence \( F(g)(m) = (m, [m(e)]) \).

It holds that SMT \( \vdash \delta'(e) = \delta'e \). Together with the soundness assumption on the SMT solver, this yields that \( m_1(e) = m_2(e) \) for all memories \( m_1, m_2 \) where \( m_1 =_L m_2 \). This also entails that \( m(e) = m_1(e) \) and hence \( \pi_2(F(g)(m)) = \pi_2(F(g)m_1) \).

From the induction hypothesis we obtain \( m_1 \in k_e(m, F(f)) \), i.e. \( \pi_2(F(f)(m)) = \pi_2(F(f)(m_1)) \). Combining the above facts yields \( \pi_2((F(g) \cap F(f))(m)) = \pi_2((F(g) \cap F(f))(m_1)) \) and hence \( m_1 \in k_e(m, F(g) \cap F(f)) \) as desired.

This allows us to prove soundness:

**Proof of Theorem 2.3.3.2:** Assume that \( \vdash_s c \) and \( (c, m_0) \xrightarrow{f_0}^* (c', m'_0) \). With Lemma 2.C.2.6, we obtain that

\[
\langle c, \delta_0, \varphi_0 \rangle \xrightarrow{f}^* \langle c', \delta', \varphi' \rangle \wedge F(f) = f_0 \wedge m_0 \vdash \varphi'
\]

With Lemma 2.C.2.7, this yields \( \forall m. [m]_L \subseteq k_e(m, F(f)) \) and hence \( [m]_L \subseteq k_e(m, f_0) \) as desired.

**Alternative treatment of while statements.** In order to avoid unrolling loops, a conservative rule such as the following can be added without compromising soundness:

\[
\frac{\text{S-WHILEINV}}{\text{SMT } \not\vdash \neg \delta'(e) \quad \langle c, \delta', \varphi \wedge \delta'(e) \rangle \leadsto \langle \varepsilon, \delta'', \varphi'' \rangle \quad \langle \text{while } e \text{ do } c, \delta, \varphi \rangle \leadsto \langle \varepsilon, \delta'', \varphi'' \rangle}
\]
where \( \delta' \) replaces all variables assigned to in \( c \) by fresh variables of the same security level. Intuitively, this “forgets” all information about variables that can be modified in the body of the loop and tries to verify the loop body. If this succeeds, the body of the while loop cannot leak any information through explicit flows.

2.D Additional Developments

2.D.1 Weak Secrecy for Machine Code

Figure 2.D.1 gives the semantics for the extracted instructions; \( i : is \) denotes a list beginning with instruction \( i \) followed by instructions \( is \). All the instruction have their usual semantics, with the exception of an added instruction \texttt{savepc pc} that stores the value \( pc \) in \( mem[sp] \).

Based on the modified machine, we define noninterference and weak secrecy:

2.D.1.1 Definition: A sequence of instructions \( is' \) satisfies noninterference with respect to original program \((is, pc_0)\), written \( NI \models_{(is, pc_0)} is' \), iff whenever

\[
\begin{align*}
& (mem_1[pc_0 \mapsto is], reg_1) = L (mem_2[pc_0 \mapsto is], reg_2) \\
& (mem_1[pc_0 \mapsto is], reg_1, is) \xrightarrow{\alpha_1} (mem_1', reg_1', \varepsilon) \\
& (mem_2[pc_0 \mapsto is], reg_2, is) \xrightarrow{\alpha_2} (mem_2', reg_2', \varepsilon)
\end{align*}
\]

it holds that \( \alpha_1 = \alpha_2 \).

2.D.1.2 Definition: A program \((is, pc_0)\) satisfies weak secrecy, written \( WS \models (is, pc_0) \), iff whenever \( (mem[pc_0 \mapsto is], reg, pc_0) \xrightarrow{is} (mem', reg', pc') \), it holds that \( NI \models_{(is, pc_0)} is' \).

As for while programs, the extracted function encodes the program that is recorded by weak secrecy and again both properties coincide. To prove this, we establish similar auxiliary lemmas as in Appendix 2.C.1
2.D.1.3 Lemma: Whenever \((\text{mem}_0[pc_0 \mapsto is], \text{reg}_0, pc_0) \xrightarrow{\alpha_0 \cdot f\text{'}, f}^*\) allowbreak(\(\text{mem}'_0, \text{reg}'_0, pc'_0\)) \& (\(\text{mem}[pc_0 \mapsto is], \text{reg}, is'\) \xrightarrow{g}^* allowbreak(\(\text{mem}', \text{reg}', \varepsilon\)), it holds that \(g = f\), where \(is'\) is the extracted program and \(f\) is the state transformer.

**Proof:** Straight-forward induction on \(\xrightarrow{\alpha_0 \cdot f\text{'}, f}^*\). □

2.D.1.4 Lemma: Whenever \((\text{mem}_0[pc_0 \mapsto is], \text{reg}, pc_0) \xrightarrow{\alpha_0 \cdot f\text{'}, f}^* (\text{mem}'_0, \text{reg}'_0, pc'_0) \& f((\text{mem}[pc_0 \mapsto is], \text{reg})) = ((\text{mem}', \text{reg)'), \alpha),\) we have \((\text{mem}[pc_0 \mapsto is], \text{reg}, is') \xrightarrow{\alpha}^* (\text{mem}', \text{reg}', \varepsilon)\).

**Proof:** By induction on \(\xrightarrow{\alpha_0 \cdot f\text{'}, f}^*\). □

2.D.1.5 Theorem: For any program \((is, pc_0)\), \(ES \models (is, pc_0)\) holds iff \(WS \models (is, pc_0)\).

**Proof:** Analogous to proof for Theorem 2.2.4.1. □
\[
\text{decode}(\text{mem}[pc]) = \text{out } r \quad \tau(r) = H
\] (T-OUTFAIL)

\[
\text{decode}(\text{mem}[pc]) = \text{out } r \quad \tau(r) = L
\] (T-OUTL)

\[
\text{decode}(\text{mem}[pc]) = \text{jal } r \quad a = \text{reg}[sp] \quad \text{mem}' = \text{mem}[a \rightarrow pc]
\] (T-JAL)

\[
\text{decode}(\text{mem}[pc]) = \text{load } r_a \ r_d \quad v = \text{mem}[\text{reg}[r_a]]
\] (T-LOAD)

\[
\text{decode}(\text{mem}[pc]) = \text{const } i \ r
\] (T-CONST)

\[
\text{decode}(\text{mem}[pc]) = \text{op } \odot r_1 \ r_2 \ r_d \quad \text{reg}' = \text{reg}[r \rightarrow \text{reg}[r_1] \oplus \text{reg}[r_2]] \quad \ell = \tau(r_1) \sqcup \tau(r_2)
\] (T-OP)

\[
\text{decode}(\text{mem}[pc]) = \text{halt}
\] (T-HALT)

\[
\text{decode}(\text{mem}[pc]) = \text{mov } r_s \ r_d
\] (T-MOV)

\[
\text{decode}(\text{mem}[pc]) = \text{store } r_a \ r_v \quad a = \text{reg}[r_a] \quad \text{mem}' = \text{mem}[a \rightarrow \text{reg}[r_v]]
\] (T-STORE)

\[
\text{decode}(\text{mem}[pc]) = \text{nop}
\] (T-NOP)

Figure 2.2: Dynamic Tainting for Machine Code
\[
\text{SMT} \vdash \text{\texttt{forget}}(\delta(e)) = \text{\texttt{forget}}(\overline{\delta(e)})
\]
\[
\langle \text{out } e, \delta, \varphi \rangle \leadsto (\varepsilon, \delta, \varphi)
\]
\[
\langle x := e, \delta, \varphi \rangle \leadsto (\varepsilon, \delta[x \mapsto \delta(e)], \varphi)
\]
\[
\langle c_1, \delta, \varphi \land \delta(e) \rangle \leadsto^* (\varepsilon, \delta_1, \varphi_1)
\]
\[
\langle c_2, \delta, \varphi \land \lnot\delta(e) \rangle \leadsto^* (\varepsilon, \delta_2, \varphi_2)
\]
\[
\text{SMT} \not\vdash \varphi \rightarrow \delta(e) \quad \text{SMT} \not\vdash \varphi \rightarrow \lnot\delta(e)
\]
\[
\langle \text{if } e \text{ then } c_1 \text{ else } c_2, \delta, \varphi \rangle \leadsto (\varepsilon, \delta_1 \oplus \delta(e), \delta_2, \varphi)
\]
\[
\langle \text{while } e \text{ do } c, \delta, \varphi \rangle \leadsto
\langle \text{if } e \text{ then } (c ; \text{while } e \text{ do } c) \text{ else } \varepsilon, \delta, \varphi \rangle
\]
\[
l \in \text{Loc fresh} \quad \delta' = \delta[x \mapsto l]
\]
\[
\langle x := \text{\texttt{new Object}}(), \delta, \varphi \rangle \leadsto (\varepsilon, \delta', \varphi)
\]
\[
l = \delta(y) \quad \delta' = \delta[x \mapsto \delta(l, f)]
\]
\[
\langle x := y.f, \delta, \varphi \rangle \leadsto (\varepsilon, \delta', \varphi)
\]
\[
l = \delta(x) \quad \delta' = \delta[(l, f) \mapsto y]
\]
\[
\langle x.f := y, \delta, \varphi \rangle \leadsto (\varepsilon, \delta', \varphi)
\]

Figure 2.3: Static Analysis for Taint Tracking
2.D. Additional Developments

\[
\text{decode}(\text{mem}[pc]) = \text{nop}
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \rightarrow (\text{mem}, \text{reg}, pc + 1)}{(E-\text{Nop})}
\]

\[
\text{decode}(\text{mem}[pc]) = \text{const } i \ r
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \xrightarrow{\text{const } i \ r} (\text{mem}, \text{reg}[r \mapsto i], pc + 1)}{(E-\text{Const})}
\]

\[
\text{decode}(\text{mem}[pc]) = \text{op} r_1 r_2 r_d
\]
\[
\frac{\text{reg}' = \text{reg}[r \mapsto \text{reg}[r_1] \oplus \text{reg}[r_2]]}{(\text{mem}, \text{reg}, pc) \xrightarrow{\text{op} r_1 r_2 r_d} (\text{mem}, \text{reg}', pc + 1)} (E-\text{Op})
\]

\[
\text{decode}(\text{mem}[pc]) = \text{mov } r_s r_d
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \xrightarrow{\text{mov } r_s r_d} (\text{mem}, \text{reg}[r_d \mapsto \text{reg}[r_s]], pc + 1)}{(E-\text{Mov})}
\]

\[
\text{decode}(\text{mem}[pc]) = \text{load } r_a r_d \quad v = \text{mem[reg}[r_a]]
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \xrightarrow{\text{load } r_a r_d} (\text{mem}, \text{reg}[r_d \mapsto v], pc + 1)}{(E-\text{Load})}
\]

\[
\text{decode}(\text{mem}[pc]) = \text{store } r_a r_v \quad a = \text{reg}[r_a]
\]
\[
\frac{\text{mem}' = \text{mem}[a \mapsto \text{reg}[r_v]]}{(\text{mem}, \text{reg}, pc) \xrightarrow{\text{store } r_a r_d} (\text{mem}', \text{reg}, pc + 1)} (E-\text{Store})
\]

\[
\text{decode}(\text{mem}[pc]) = \text{out } r
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \xrightarrow{\text{out } r} (\text{mem}, \text{reg}, pc + 1)}{(E-\text{Out})}
\]

\[
\text{decode}(\text{mem}[pc]) = \text{jump } r
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \rightarrow (\text{mem}, \text{reg}, \text{reg}[r])}{(E-\text{Jump})}
\]

\[
\text{decode}(\text{mem}[pc]) = \text{jal } r
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \xrightarrow{\text{savepc } pc} (\text{mem}[\text{reg}[sp] \mapsto pc], \text{reg}, \text{reg}[r])}{(E-\text{Jal})}
\]

\[
\text{decode}(\text{mem}[pc]) = \text{bnz } i \ r \quad \text{reg}[r] \neq 0
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \rightarrow (\text{mem}, \text{reg}, i)}{(E-\text{BnzTrue})}
\]

\[
\text{decode}(\text{mem}[pc]) = \text{bnz } i \ r \quad \text{reg}[r] = 0
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \rightarrow (\text{mem}, \text{reg}, pc + 1)}{(E-\text{BnzFalse})}
\]

\[
\text{decode}(\text{mem}[pc]) = \text{halt}
\]
\[
\frac{(\text{mem}, \text{reg}, pc) \rightarrow (\text{mem}, \text{reg}, \varepsilon)}{(E-\text{Halt})}
\]
\[
i = \text{nop} \\
\text{W-Nop}
\]
\[
(mem, reg, i : is) \rightarrow (mem, reg, is) \\
\]
\[
i = \text{const } i r \\
\text{W-Const}
\]
\[
(mem, reg, i : is) \rightarrow (mem, reg[r \mapsto i], is) \\
\]
\[
i = \text{op}_\oplus r_1 r_2 r_d \\
\text{W-Op}
\]
\[
reg' = \text{reg}[r \mapsto \text{reg}[r_1] \oplus \text{reg}[r_2]] \\
(mem, reg, i : is) \rightarrow (mem, reg', is) \\
\]
\[
i = \text{mov } r_s r_d \\
\text{W-Mov}
\]
\[
(mem, reg, i : is) \rightarrow (mem, reg[r_d \mapsto \text{reg}[r_s]], is) \\
\]
\[
i = \text{load } r_a r_d \\
\text{W-Load}
\]
\[
v = \text{mem}[\text{reg}[r_a]] \\
(mem, reg, i : is) \rightarrow (mem, reg[r_d \mapsto v], is) \\
\]
\[
i = \text{store } r_a r_v \\
\text{W-Store}
\]
\[
a = \text{reg}[r_a] \\
mem' = \text{mem}[a \mapsto \text{reg}[r_v]] \\
(mem, reg, i : is) \rightarrow (mem', reg, is) \\
\]
\[
i = \text{out } r \\
\text{W-Out}
\]
\[
\text{reg}[r] \rightarrow (mem, reg, is) \\
\]
\[
i = \text{savepc } pc \\
\text{W-SavePC}
\]
\[
(mem, reg, i : is) \rightarrow (mem[sp \mapsto pc], reg, is) \\
\]
CHAPTER
THREE

JSLINQ: Building Secure Applications across Tiers
Musard Balliu, Benjamin Liebe,
Daniel Schoepe, Andrei Sabelfeld

Abstract. Modern web and mobile applications are complex entities amalgamating different languages, components, and platforms. The rich features span the application tiers and components, some from third parties, and require substantial efforts to ensure that the insecurity of a single component does not render the entire system insecure. As of today, the majority of the known approaches fall short of ensuring security across tiers.

This paper proposes a framework for end-to-end security, by tracking information flow through the client, server, and underlying database. The framework utilizes homogeneous meta-programming to provide a uniform language for programming different components. We leverage .NET meta-programming capabilities from the F# language, thus enabling language-integrated queries on databases and interoperable heterogeneous execution on the client and the server. We develop a core of our security enforcement in the form of a security type system for a functional language with mutable store and prove it sound. Based on the core, we develop JSLINQ, an extension of the WebSharper library to track information flow. We demonstrate the capabilities of JSLINQ on the case studies of a password meter, two location-based services, a movie rental database, an online Battleship game, and a friend finder app. Our experiments indicate that JSLINQ is practical for implementing high-assurance web and mobile applications.
3.1 Introduction

There is no such thing as a free lunch - building secure and robust web applications is a complex and error prone task. A recurrent fact attested by investigations from security organizations and communities of security experts [8, 4], and very frequently reported by the media [9, 7], is that vulnerabilities in web and mobile applications dominate the classifications of the most dangerous security attacks. The reason can be attributed to different factors, including the myriad of programming languages, technologies and platforms which are used to build modern applications. This process requires substantial efforts and skills on the programmer’s side for getting the application logic right, let alone secure and reliable. In this paper, we set out to study the challenge of heterogeneity and provide practical solutions with formal evidence, that help a programmer to build web and mobile applications in a secure manner. In particular, we focus on vulnerabilities that go beyond injection attacks and affect the business logic of the entire web application.

A closer look at a typical web architecture shows that web applications are often distributed over several tiers: (a) a client tier, where most of the UI logic runs in a web browser as JavaScript and HTML including third-party libraries; (b) a server tier, where the bulk of the application logic is executed in a language like F#, Java or other; (c) and a database tier that serves as persistent store and executes e.g. SQL code. Common security attacks rely on the fact that applications are implemented in different languages that span tiers with different trust relationships. As a result, many security policies are application-specific and tightly connected to the application logic and the trust relationships between the involved parties.

Motivating Scenarios: The following scenarios illustrate the need for cross-tier security analysis and policies.

Password Meter: The first scenario considers a client-side password meter, which is a program used to estimate the strength of passwords provided by users. It is important that the chosen password is not leaked to an application server or other third parties. A reasonable security policy treats the password field as sensitive, and the third-party and the RPC functions used to communicate with the application as public, while enforcing that no sensitive
information flows to the public destinations.

*Location-based Service:* The second scenario is a location-based service, which uses location information to query a web service for the list of nearby points of interest, and a third-party map library to display these points. However, users concerned about privacy may not want to reveal the exact coordinates of their location. A reasonable security policy allows for a declassification function to obfuscate the real location, and only send approximate coordinates to the location server. Moreover, the map library should only be used to display the points of interest and not to, for instance, leak the browser’s cookie to the library provider.

*Friend Finder App:* The third scenario is a mobile app. The user wants to know if a friend is using a certain app, say WhatsApp, without revealing the friend’s phone number to the remote server in case they are not using that app. This can be avoided by using a hash function to hide the phone number before sending it to the database server, which in turn compares the hashed value to the list of its users’ phone numbers and replies whether or not that user is using the app. A reasonable policy considers the phone address book as sensitive and ensures that only hashed values are sent to the untrusted application server for discovery.

These are all examples of how a security attack can occur across all three tiers of an application. Hence, a satisfactory security analysis needs to express and validate policies for applications that span client, server and database tiers.

**Attacker Model:** Different attacker models arise in multi-tier web applications. Sensitive or untrusted data may originate from any of the components, for instance it can be a user location from the client, a password from the database or an authentication key from the server. Consequently, any tier can be subject to unintentional or malicious information leaks toward another tier. The policies for the first two scenarios constrain the sensitive data of a trusted client wrt. an untrusted third-party library and a (partially) trusted server. The third scenario illustrates policies for a trusted client wrt. to a completely untrusted server. The client can also be untrusted. For example a trusted server, after authenticating a user, may read his personal data from a trusted database and send back a customized web page, however, no information about other users in the database should flow to the client. Meaningful
combinations of tiers and attacker models will be discussed in Section 3.4. We do not address network attackers who intercept, alter or deny communication between tiers, while techniques like SSL can be used to prevent these types of attacks.

**State of the Art:** Information-flow control (IFC) tracks sensitive (untrusted) data throughout the computation ensuring that no illegal information flows from sensitive (untrusted) sources toward public (trusted) sinks. This provides end-to-end security guarantees as required in the scenarios above. In general, we mark sources and sinks with labels from a lattice of security levels that expresses the trust relationships between parties. E.g., horizontal privilege-escalation attacks can be prevented by assigning separate security labels for separate users. A large body of work has studied dynamic and static enforcement techniques for all levels of the hardware and software stack [22, 34], including web applications [26] and distributed systems [42]. The majority of these works tackles the problem of information flow for different components in isolation [38, 29, 23]. This is unsatisfactory because tracking information across tiers is necessary for end-to-end security. A few works, as discussed in Section 3.5, bridge IFC across components allowing for policies that regulate information flows for a web application as a whole. Noteworthy, recent frameworks integrate database queries into programming languages for client and server applications providing a uniform way to program an entire web application, including reasoning about security [18, 16, 15].

**Contributions:** In this paper we leverage homogeneous metaprogramming to obtain a uniform language for reasoning about web and mobile application security across the client-server-database boundaries. The .NET facilities provide support for language-integrated queries on databases and interoperable heterogeneous execution for client and server applications, embedding them seamlessly in the F# language [40]. This allows to implement an entire web or mobile application as a simple F# program and then let the compiler split the code transparently for each tier. In this work we enrich a subset of the language with security types which allow to express security policies. We implement the security types by custom attributes as a separate F# module on top of existing fully-fledged development in F#, providing a complete separation between the program code and the security policy. We then exe-
cute the security type check as a separate verification step followed by the F# compilation and thus leaving the F# type system untouched. Finally, we split the program into three parts, producing JavaScript and HTML code to run on the browser, SQL code to run on the database and F# code to run on the server.

On the formal side (Section 3.2), we develop a model for a functional language with references (a subset of F#), quotations and antiquotations, and establish the soundness of the security type system. Our soundness proof extends and generalizes the proof technique introduced by Pottier and Simonet [30] with support for arbitrary data types and declassification policies. The query language is based on the one introduced by Cheney et al. [14] and uses quotation and normalization of quoted terms to model the semantics of the database language. For simplicity, our results assume a two-point security lattice for confidentiality, however, they apply to arbitrary lattices, including integrity, in a similar fashion.

On the practical side (Section 3.3), we have implemented JSLINQ, an extension of WebSharper [10] and LINQ [1] libraries with IFC. With JSLINQ, a developer can use a fully-fledged language such as F# for writing secure web and mobile applications. A security analyst is expected to know what sources and sinks are sensitive, which is a reasonable assumption so long as they are partially trusted. If the developer is malicious, one can leverage techniques from [27, 31] to automatically extract sources and sinks used by the application (this is out of scope in this work). The policy module requires to specify security signatures once and only for the APIs that are actually used, thus making it easier and less time-consuming for the programmer. Our experience shows that JSLINQ provides a good trade-off between annotation burden and security assurance for developers with some security background, while user studies with non-expert developers are subject to future work.

We demonstrate the capabilities of JSLINQ on several realistic case studies (Section 3.4), including the scenarios discussed above, a password meter and an online Battleship game. The case studies leave out user interfaces and other boilerplate code, and only focus on the security-critical parts of the applications to demonstrate the potential of our technique. Moreover, compositionality of the security type checking makes the approach scalable to arbitrary lines
of code. The experiments show that JSLINQ is useful for building secure applications and it enjoys several advantages compared to existing tools (Section 3.5 and Table 3.2).

A precursor of our approach is SELINQ by Schoepe et al. [36]. SELINQ uses a security type system to enforce policies for server-database applications written in F#, as we do. Rather than enriching F# with security types, SELINQ implements a subset of the language presented in Section 3.2 and uses a compiler implemented in Haskell to type check and generate F# executable code. By contrast, JSLINQ closes the end-to-end loop by supporting client-side, including third-party code, for fully-fledged F# applications. A distinguishing feature of JSLINQ is that security type checking does not interfere with the normal development process. In practical terms, this translates to a big gain as the programmers can use a production-grade system to develop applications, yet leverage a security type system to verify the critical parts of the code. Moreover, practicality of JSLINQ is supported by several case studies and security policies. Declassification allows us to handle richer policies, e.g. only friends can view a user’s profile data, while dynamic policies would require extending the type system with techniques from [43]. While both SELINQ and JSLINQ use the framework by Cheney et al. [14], JSLINQ significantly extends that formalism with mutable references and declassification using a different technique to show noninterference.

While our main focus is on multi-tier application-level attacks, JSLINQ inherits protection against XSS and SQL injection attacks from its components, respectively, from WebSharper and LINQ. Such attacks are impossible due to strong typing [32], similar to frameworks as GWT. For instance, an SQL injections are prevented by the use of LINQ, which leverages the underlying F# type system to strongly type all database queries.

The full details of the framework, including semantics and proofs, and the code for JSLINQ are available online [11].

3.2 Framework

In this section we present the formal underpinnings of the framework. The client and the server components are written in the host language, while the database component is written in the quoted
language. The framework consists of a functional language with mutable storage and support for product types, records, lists, quotations and antiquotations, the security type system, and shows that the type system enforces noninterference and declassification policies with respect to the operational semantics. The host and the quoted language represent a core of the F# language as implemented by JSLINQ.

### 3.2.1 Language

The language is presented in Figure 3.1. It includes the usual constructs of a functional language with references, extended with quotations and antiquotations to account for database queries. The syntax consists of security levels, types, and terms. \( \mathcal{T} \) denotes a sequence of entities \( x \).

\[
\ell ::= \text{L} | \text{H} \quad \text{(security types)}
\]

\[
b ::= \text{bool}^\ell | \text{int}^\ell | \text{float}^\ell | \text{string}^\ell \quad \text{(base types)}
\]

\[
t ::= b | \text{unit} | t \rightarrow t | t \text{ref}^\ell | t \ast t | \{ \overline{f : t} \} | (t \text{ list})^\ell | \text{Expr}(t) \quad \text{(general types)}
\]

\[
T ::= (\{ \overline{f : b} \}) \text{ list}^\ell \quad \text{(database tables)}
\]

\[
\Gamma, \Delta, M ::= \cdot | \Gamma, x : t | \Delta, x : t | M, l : t \quad \text{(type environment)}
\]

\[
e ::= () | c | x | l | op(\overline{e}) | \text{lift} \ e | \text{fun}(x) \rightarrow e \quad \text{(terms)}
\]

\[
| \text{rec} \ f(x) \rightarrow e \ | (e, e) | \text{fst} \ e | \text{snd} \ e | \{ \overline{f = e} \} | e.f
\]

\[
| \text{yield} \ e | [] | e @ e | \text{for} \ x \ \text{in} \ e \ \text{do} \ e | \text{exists} \ e
\]

\[
| \text{if} \ e \ \text{then} \ e \ \text{else} \ e | \text{if} \ e \ \text{then} \ e | \text{run} \ e | <@ \ e @> | \% \ e)
\]

\[
| \text{database}(x) | \text{ref} \ e | \!e | e := e
\]

Figure 3.1: Syntax of language and types

We remark on some of the interesting constructs: \( c \) denotes built-in constants, such as booleans, integers, floats and strings. \( op \) denotes built-in operators, such as addition and logical connectives. \( \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 \) evaluates to \( e_2 \) if \( e_1 \) evaluates to \text{true} and to \( e_3 \) otherwise. The language includes mutable state. Terms \( \text{ref} \ e \) (reference creation), \!\( e \) (dereference) and \( e := e \) (assignment) denote, respectively, allocating, dereferencing and updating mem-

ory locations. () denotes a value of type \texttt{unit}. Database queries are modeled by quoted expressions \( \langle @ e @ \rangle \) of type \texttt{Expr}(t). The language allows only closed quoted terms, since this simplifies the semantics of the language and is still able to express all the desired concepts. Quoted functions can be expressed by abstracting in the quoted term as opposed to abstracting on the level of the host language. \( \langle \% e \rangle \) denotes antiquotation of the expression \( e \), and allows splicing of quoted expressions into quoted expressions in a type-safe way. \texttt{lift} \( e \) lifts an expression of type \( t \) to type \texttt{Expr}(t). \texttt{for} \( x \) in \( e_1 \) do \( e_2 \) is used to express list comprehensions where \( x \) is bound successively to elements in \( e_1 \) when evaluating \( e_2 \). The results of evaluating \( e_2 \) for each element are then concatenated. \texttt{run} \( e \) denotes running a quoted expression \( e \), which involves generating an SQL query based on the quoted term. \( e_1 \odot e_2 \) denotes concatenation of \( e_1 \) and \( e_2 \). \texttt{exists} \( e \) evaluates to \texttt{true} if and only if the expression \( e \) does not evaluate to the empty list. This can be used to check if the result of a query is empty. \texttt{if} \( e_1 \) then \( e_2 \) evaluates to \( e_2 \) if \( e_1 \) evaluates to a non-empty list and to \texttt{[]} otherwise. \texttt{yield} \( e \) denotes a singleton list consisting of expression \( e \).

\textbf{Security type language:} Security types are defined by annotating a standard type language for a functional fragment with quotations and references with security levels \( \ell \). The security levels are taken from the two-element lattice \( \langle \{ L, H \}, \sqsubseteq \rangle \) consisting of a level \( L \) for low-confidentiality (dually high-integrity) information and a level \( H \) for high-confidentiality (dually low-integrity) information. The ordering relation requires that \( L \sqsubseteq H \). The types are split into base types \( (b) \), which can occur as types of columns in tables \( (T) \), and general types \( (t) \) which include unit, functions, references, tuples, records, lists, and quoted expression types. Function types include a level \( \ell \), which is a lower bound on the level of locations that might be written to when the function is called. To avoid such leakages the function is only allowed to write to memory cells with security levels greater than \( \ell \). Reference types \( t \ \texttt{ref}^{\ell} \), besides the security level \( t \) of the value stored at the associated location, carry a level \( \ell \) which represents the security level of the reference itself. This is because references are themselves first-class values and can hence be used to leak confidential information.

As is common, a database is a collection of tables. Each table consists of at least one named column, each of which equipped with
a fixed security type. The security levels on types for database columns express the confidentiality of the data contained in that column. In particular, each database is given a type signature $\Sigma$ to express security policies for databases. A type signature describes tables as lists of records. Each record field corresponds to a column in the sense that the field name matches the name of the column in the database. The security level of a column is specified by using a suitable type for the corresponding field in the record. The ordering of elements in a list is irrelevant.

Types are equipped with a subtyping relation $\sqsubseteq$, which is an extension of the lattice ordering relation. The subtyping relation is standard [30, 24], therefore we do not report it here. With a little abuse of notation, we use the subtyping relation to compare security annotations $\ell$ with types $t$. In particular, if the type carries a security annotation $\ell'$, we compare the security levels $\ell \sqsubseteq \ell'$. Otherwise, we need to open the type and look inside the type constructor as described in Figure 3.2.

\[
\begin{align*}
\frac{\ell \sqsubseteq \ell'}{\ell \sqsubseteq t'} & \quad \frac{\ell \sqsubseteq \text{pc} \quad \ell \sqsubseteq t}{\ell \sqsubseteq t'} & \quad \frac{\ell \sqsubseteq t_1 \quad \ell \sqsubseteq t_2}{\ell \sqsubseteq t_1 \cdot t_2} \\
{\ell \sqsubseteq \text{unit}} & \quad \frac{\ell \sqsubseteq t \quad \ell \sqsubseteq t'}{\ell \sqsubseteq \text{pc}} & \quad \frac{\ell \sqsubseteq t}{\ell \sqsubseteq \text{Expr}\langle t \rangle}
\end{align*}
\]

Figure 3.2: Security annotation constraints

To illustrate the addition of security levels to the type system in the case of multi-tier applications, consider an example involving a database of people locations and friends, LocationDB. The locations are confidential, while the names are not, which leads to the following type for LocationDB.

LocationDB :
{ People :
  { Id : int^L; Name : string^L;
    Lon : float^H; Lat : float^H } list^L
; Friends :
  { Id1 : int^L ; Id2 : int^L } list^L
}

Suppose John wants to know whether there are any friends within the range of 1km from his current location. We can query
the database for the list of John’s friends and later calculate the distance relative to John’s location. This can be done by iterating once over all friends in the database to retrieve the list of John’s friends and twice over all people in the database to retrieve the result information. After finding John’s Id in the database, we check that whenever it occurs in the Friends table as Id1, the corresponding friend as Id2 occurs in the People table as Id. In that case, the name, the latitude and the longitude of that friend is returned as part of the result.

```fsharp
let db = <@ database "LocationDB" @>

type ResultType =
    { name:string^L; lon:float^H; lat:float^H }

let friendsLoc : Expr < ResultType list^L > =
    <@ for f in (% db).Friends do
        for p1 in (% db).People do
            for p2 in (% db).People do
                if (p1.Name = "John") && (p1.Id = f.Id1) &&
                    (f.Id2 = p2.Id) then
                    yield ({ name = p2.Name; lon = p2.Lon
                            ; lat = p2.Lat })
    @>
```

The information flow policy for the program is specified by giving a type annotation to the quoted expression that generates the query, i.e., a type annotation for `friendsLoc`. In particular the name component of the result is public, while the location information is confidential as described by `ResultType`. This matches the policy specified for the database contents, i.e., LocationDB, in which the name of people are public while their locations are not. Changing the security annotation of the name field from public to confidential should result in a type error, since the security level of the Name field of the result is public. The example so far illustrates secure information flows from the database to the server for an attacker model where the server is untrusted.

The server uses the result of the database query to calculate the distance between John’s location and his friends location, and then send to John the list of nearby friends. The function `dist : (float^ℓ ∗ float^ℓ) ∗ (float^ℓ ∗ float^ℓ) ℓ′ → float^ℓ` is side-effect free and it computes the Euclidean distance between two points. The security
The function `friendNames` takes as input a public location `publicLoc`, executes the query represented by the function `friendsLoc` on the database and returns a list of public names of nearby friends. Since the location information contained in the result of `friendsLoc` is confidential, there is an implicit flow from the location to the list of names. In fact, a public observer learns that the location of everyone in the returned list of names is within 1km from the location `publicLoc`. Therefore, the security type checking should fail. However, one may consider acceptable to leak the distance information as long as the exact location is protected. This can be achieved by *declassifying* the function `dist`, i.e., considering its result as public, although part of the input is confidential. At last, John can call the remote function `friendNames` on the client-side by providing his current location `locJohn`.

```
let locJohn : (float*L, float*L) = GetLocation()

let friends : string*L list*L = friendNames locJohn
```

The function is executed on the server-side and it interacts with the database to retrieve information as described above. Then the list of names of nearby friends is returned back to John on the client-side. The security type checker will ensure that there are no insecure information flows, except the allowed ones, from the database to the client.

### 3.2.2 Operational Semantics

The operational semantics of the language evaluates terms in the context of a mutable store $\mu$ and a database $\Omega$. A partial mapping $\mu : \text{Loc} \rightarrow \text{Val}$ from locations to values models the semantics of memory effects. We write $\mu[l \mapsto v]$ for a store $\mu$ which maps location $l$ to value $v$, otherwise agrees with $\mu$. A *configuration* $(e, \mu)$
is a pair of a term $e$ and a store $\mu$. We write $e$ when $\mu$ is empty. We denote evaluation of a configuration $(e, \mu)$ using database data in $\Omega$ to another configuration $(e', \mu')$ by $(e, \mu) \xrightarrow{\Omega} (e', \mu')$. $\Omega$ is a function that maps database names to the actual content of the database it refers to, and $\delta$ is a function that maps operators to their corresponding semantics. $\Sigma$ maps constants and databases to their respective types. We assume that $\Omega$ is consistent with the typing for databases given in $\Sigma$: for each database $\Omega(db)$ is assumed to be a value of type $\Sigma(db)$. Let $\xrightarrow{\Omega}^*$ be the reflexive-transitive closure of $\xrightarrow{\Omega}$. Evaluation and normalization of the quoted language is denoted by $\text{eval}_\Omega(\text{norm}(e))$. Figure 3.7 shows the syntax of normalized terms. This evaluation generates database queries that can be translated to SQL and executed by actual database servers. For instance, higher-order features such as nested records or function applications need to be evaluated to obtain computations that can be expressed in SQL. The syntax of values and evaluation contexts can be defined both for the host language and the quoted language as described in Figure 3.8. The quoted language is purely functional and contains no recursion. The evaluation contexts ensure that the semantics is call-by-value with left-to-right evaluation of terms. Quotation contexts $Q$ are used to ensure that there are no antiquotations left of the hole. The evaluation rules for the host language are standard as reported in Figure 3.9. We denote the substitution of free occurrences of variable $x$ in term $e$ with another term $e'$ by $e[x \mapsto e']$. The evaluation contexts entail sequentiality and let-binding between terms; we write $e_1; e_2$ for $(\text{fun}(x) \rightarrow e_2)e_1$, where $x$ is not free in $e_2$ and $\text{let } x = e \text{ in } e'$ for $(\text{fun}(x) \rightarrow e')e$. The evaluation rules for the query language, as presented in Figures 3.10 and 3.11, follow Cheney et al. [14].

To avoid clutter, we omit the store component from configurations since the quoted language is purely functional.

### 3.2.3 Security Condition

The security condition expresses the notion of noninterference for a functional language with references and databases. Noninterference is an information flow policy that formalizes computational independence between confidential and public information, guaranteeing that no information about the former can be inferred from the latter. More precisely, this is expressed as the preservation of
an equivalence relation under pairwise execution; given two inputs
that are equal in the components that are visible to an attacker,
evaluation should result in two output values that also coincide
in the components that can be observed by the attacker. Memory
locations are not directly observable by the attacker, however their contents may affect the output returned by the computations and thus leak information. For example, the program

\textbf{let} \( l = \text{ref true} \text{H in } !l \) uses a public location \( l \), which stores a confidential value \textbf{true}, to leak that value to an attacker through the dereference \( !l \).

To establish the behavior of a secure program from the perspective of an attacker, we introduce the notion of low-equivalence denoted by \( \sim \) that demands that parts of values with types that are annotated with \( L \) are equal, while placing no demands on the high counterparts. Low-equivalence is formalized as a family of equivalence relations \( \sim_t \) on values parametrized by types. We omit the subscript on \( \sim \) when the type is clear from the context and write \( \sim \) for sequences of values. To present the relations in a more concise manner, we combine the cases for different security levels using implication in the premises; e.g. equality on base types is only required if the security level is \( L \).

\textbf{Definition 1} \( (\sim_t) \). \textit{The family of equivalence relations \( \sim_t \) is defined inductively by the rules in Figure 3.3.}

Built-in values \( c \) of base type \( b \) are compared using equality if the values are public. Unit values \( () \) are related by \( \sim_{\text{unit}} \) and do not contain security levels. In the case of function types and quoted expressions, \( \sim_t \) corresponds to noninterference for the bodies of the functions. Moreover, functions are related by \( \sim_t \ell \to t' \) if for all input values related by \( \sim_t \) they evaluate to values related by \( \sim_{t'} \) and the memory effects are upper bounded by the security level of the result \( \ell \subseteq t' \). Records are related by \( \sim \) if they contain the same fields, and each field’s contents are also related by \( \sim \). Similarly, tuples are related by \( \sim \) if the corresponding components are related by \( \sim \). Two lists are required to have the same length if the list type is annotated with \( L \), but their contents may differ based on the element type. To illustrate this, consider two lists of integers \( l_1 = \text{yield } 1 \text{ @ } [] \) and \( l_2 = \text{yield } 2 \text{ @ } [] \). If the lists are typed with the type \( t = (\text{int}^H \text{ list})^L \), the length of the list is considered
\[
\ell = L \Rightarrow c' = c'' \\
\frac{c' \sim_{\ell} c''}{\text{fun}(x) \rightarrow e_1 \sim_{t_{\text{ref}_t}} \text{fun}(x) \rightarrow e_2}
\]

\[
\ell = L \Rightarrow l_1 = l_2 \\
\frac{l_1 \sim_{t_{\text{ref}}} l_2}{\text{rec } f(x) \rightarrow e_1 \sim_{t_{\text{ref}_t}} \text{rec } f(x) \rightarrow e_2}
\]

\[
\frac{\forall v_1, v_2, v'_1, v'_2, \Omega_1, \Omega_2, \Omega_1 \sim_{\Sigma} \Omega_2 \land v_1 \sim_{t} v_2 \land \quad e_1[f \mapsto \text{rec } f(x) \rightarrow e_1, x \mapsto v_1] \rightarrow_{\Omega_1} v'_1 \land \quad e_2[f \mapsto \text{rec } f(x) \rightarrow e_2, x \mapsto v_2] \rightarrow_{\Omega_2} v'_2 \Rightarrow \\
v'_1 \sim_{t'} v'_2 \land pc \subseteq t'}
\]

\[
\{f = v\} \sim_{\langle f, l \rangle} \{f = w\} \\
\frac{v_1 \sim_{t_1} v'_1 \quad v_2 \sim_{t_2} v'_2}{(v_1, v_2) \sim_{t_1 \cdot t_2} (v'_1, v'_2)} \\
\forall \Omega_1, \Omega_2, \Omega_i \sim \Omega_2 \Rightarrow \\
eval_{\Omega_1}(\text{norm}(e_1)) \sim_{t} eval_{\Omega_2}(\text{norm}(e_2)) \\
() \sim_{\text{unit}} () \quad \quad e_1 \sim_{\text{Expr}(t)} e_2
\]

Figure 3.3: Rules for \(\sim_t\)

public, while the contents are confidential. If in contrast the type is \(t' = (\text{int} \uparrow \text{list})^L\), neither the contents nor the length of the list is confidential. Hence \(l_1 \sim_t l_2\) holds while \(l_1 \sim_{t'} l_2\) does not. Memory locations are compared using equality if the locations are public.

With this we are ready to define the top-level notion of security based on noninterference [20]. Since the family of low-equivalence relations is parametrized by types the definition is done with respect to the initial host type, the initial database type and the final result type.

**Definition 2** \((NI(e_1, e_2)_{t, \Sigma, v})\). Two expressions \(e_1\) and \(e_2\) are non-
interfering with respect to the host type \( t \), the database type \( \Sigma \) and the final type \( t' \) if for all \( \Omega_i, v_i, v'_i \) and \( \mu_i \) such that \( v_1 \sim_t v_2, \Omega_1 \sim_\Sigma \Omega_2, \) and \( e_i[x \mapsto v_i] \longrightarrow^{\star}_{\Omega_i} (v'_i, \mu_i) \) for \( i \in \{1, 2\} \) it holds that \( v'_1 \sim_\nu v'_2 \).

Given an open expression \( e \), \( NI(e, e)_{t, \Sigma, t'} \) should be read as \( e \) is secure with respect to the security policy expressed by \( t, \Sigma \) and \( t' \), i.e., no secret parts of host and the database as defined, respectively, by \( t \) and \( \Sigma \) is able to influence the public parts of the result value as defined by \( t' \). Note that the definition can represent expressions with multiple inputs by using record values. Moreover, the noninterference policy is termination-insensitive [41, 34], namely it ignores leaks via the observation of (non)termination.

**Declassification:** Noninterference is overly restrictive for programs that leak confidential information in a controlled manner, as shown by the example in Section 3.2.1. To account for these cases, we extend the framework with support for declassification policies that regulate what information can be released by the program. The policies are expressed in terms of escape hatches from a set \( D = \{d_1, \cdots, d_k\} \) and correspond to the What dimension in [35]. Escape hatches were introduced to express a similar notion, called delimited release, for imperative languages [33]. The security condition is then refined to also take into account the equivalence between declassification expressions. This requires to extend the low-equivalence relations used for noninterference with declassification.

**Definition 3** \((DNI(e_1, e_2)_{D, t, \Sigma, t'})\). Two expressions \( e_1 \) and \( e_2 \) are noninterfering with respect to the declassification expressions \( D \), the host type \( t \), the database type \( \Sigma \) and the final type \( t' \) if for all \( \Omega_i, v_i, v'_i \) and \( \mu_i \) such that \( v_1 \sim_t v_2, \Omega_1 \sim_\Sigma \Omega_2, d_j[x \mapsto v_1] \sim_{t, \Sigma} d_j[x \mapsto v_2] \) and \( e_i \longrightarrow^{\star}_{\Omega_i} (v_i, \mu_i) \) for \( i \in \{1, 2\} \) it holds that \( v'_1 \sim_\nu v'_2 \).

### 3.2.4 Security Type System

The goal of the security type system is to enforce the notion of noninterference for a functional language with references and databases. We present the typing rules for the host language in Figure 3.3. Typing judgments are of the form \( pc, \Gamma, M \vdash e : t \) where \( pc \) is the program counter level, \( \Gamma \) is a typing context mapping variables to types, \( M \) is a typing context mapping locations to types, \( e \) is an
expression and \( t \) is a type. They denote that expression \( e \) has type \( t \) in context \( pc, \Gamma, M \). We also write \( H \) for \( pc, \Gamma, M \). Intuitively, the program counter level approximates the information that can be learned by observing that the program has reached a particular point during the execution and it is used to control implicit flows due to branching on high values. For uniformity, we write \( pc, \Gamma, M \vdash v : t \) for typing judgments dealing with values, although \( pc \) is redundant given that values have no computational effects. \( \ell \sqcup \ell' \) denotes the join of levels \( \ell \) and \( \ell' \), i.e., \( \ell \sqcup \ell' = H \) iff \( H \in \{ \ell, \ell' \} \), and \( \ell \sqcup \ell' = L \) otherwise.

The typing rules for the quoted language are similar to those for the host language and are reported in Figure 3.4. Typing judgments have the form \( H, \Delta \vdash e : t \), where \( H \) is the typing context for the host language and \( \Delta \) is the typing context for the quoted language. We use the suffix \( Q \) to refer to the rule for the quoted language.

Most types contain a level \( \ell \) that denotes whether the “structure” of the value is confidential. In the case of base types, this means that their values are confidential or not. In the case of \((t \text{ list})^\ell\), the level \( \ell \) indicates whether the length of the list is confidential. If \( \ell = H \), the entire list is considered a secret, otherwise the length of the list may be disclosed to a public observer. However, the elements of the list may or may not be confidential depending on the level of the elements given by the type \( t \). Record types, pair types and quotation types do not carry an explicit level annotation, since their security level is contained in the type components. In the case of records and pairs, it suffices to annotate the type of each component, since the structure cannot be modified dynamically. For types for quoted expressions, the security annotation is contained in the type \( t \). Function types contain the usual input and output types together with a security level \( pc \) which represents a lower bound on the security level of locations that may be written when calling the function. In order to securely call the function in a context \( pc' \) it must be the case that \( pc' \sqsubseteq pc \). The intuition is that, in the presence of side-effects, the function can disclose information via its result or via its side-effects. We assume that types for operators, constants, and databases are given by the mapping \( \Sigma \). Moreover, we also assume that each query only uses a single database. Expressions in the host language differ from ex-
pressions in the quoted language. Recursion, quotation, branching (rule If) and memory operations (reference creation, dereferencing and update) are only allowed in the host language; expressions of the form `database(x)` and antiquotations are only allowed in the quoted language.

We now comment on a few typing rules. Rules `Var`, `VarQ` and `Loc` assign types to variables and locations by looking up the corresponding environment. `Fun` and `Rec` use the program counter level appearing in the functions type to check the respective function bodies. ` Apply` is used to check function application. The rule ensures that the side-effects `pc'` of the caller function are not visible in contexts for which the program counter level is `pc`, namely `pc ⊑ pc'`. As a result, it prevents a function to write to low memory locations in a high context and thus leak information through implicit flows. `Ref` checks memory allocation operations. It ensures that a low reference is not created in a high context and that it does not contain a high value. `Deref` checks dereference operations and ensures that the reference level is upper bounded by the level of its contents to avoid information leakage through aliases. `Assn` checks memory updates and ensures that no low memory writes occur in a high context or in a high location. The following example captures the intuition behind the typing rules for mutable storage. Let `l`, `l'` be variables of type `int^L ref^H`, `l''` of type `int^H ref^H` and `h` of type `bool^H`. The program is insecure since the returned value at location `l` reveals the initial value of variable `h` through aliasing.

```plaintext
l = ref 0; l' = ref 1; let l'' = if h then l else l' in l'' := 2; !l
```

The program is correctly rejected by the type system. By rule `Ref` the first two references are typable for `pc = H`. The conditional is also typable by rule `If`, since `l` and `l'` are high references. The successive assignment is typable by rule `Assn` provided that `2` has type `int^H`. The type checking fails when considering the dereference `!l`, since the rule `Deref` requires `ℓ ⊑ t`, which is not true for `l` of type `int^L ref^H`.

Lists can be assigned an arbitrary level when constructed using `yield` and `[]`. Expressions of the form `e_1 @ e_2` reveal information about the structure of both lists and hence their security levels are combined in the result type. Similarly, `exists` only reveals
information about the structure of the list, but nothing about the contents. Therefore, the security level of list contents is discarded and only the security level of the list itself is present in the result type. Rule QUOTE ensures that its arguments are typed in an empty context for quoted expressions. This expresses that only closed quoted terms are allowed in this language. Running a quoted expression $e$ of type $\text{Expr}(t)$ using $\text{run} \ e$ results in an expression of type $t$ (rule RUN). Expressions for $\text{database}(db)$ get their type from the mapping $\Sigma$. Rule ANTIQUOTE allows to entities defined in the host language from within a quoted expression. The argument of an antiquotation must itself be a quoted expression. Rules SUB and SUBQ allows raising the security level of an expression.

To illustrate the type system further, we explain the typing rule FOR rule in greater detail. Recall that for expressions are used to denote list comprehensions. The typing rule assigns the resulting list the join of the security level of both sub-expressions. The following example demonstrates why this is required.

Consider the program for $x$ in $xs$ do $ys$ that uses a for expression to leak the structure of the lists $xs$ and $ys$. We assume $xs$ to have type $(t \ \text{list})^\ell$ for some type $t$ and level $\ell$, whereas $ys$ has type $(t' \ \text{list})^{\ell'}$. Since the resulting lists for each element of $xs$ will be concatenated, the resulting list will have length $|xs| \times |ys|$, where $|a|$ denotes the length of $a$. If either $xs$ or $ys$ contains only one element, the length of the other list is revealed through the result. To account for this information flow, the resulting list will be typed with level $\ell \sqcup \ell'$.

### 3.2.5 Soundness

The soundness result is stated as the preservation of a low-equivalence relation under pairwise execution. If we start out in any two low-equivalent environments then the result of running a well-typed program will be low-equivalent with respect to the type of the program. Assuming that the typing of the execution environment corresponds to the capabilities of the attacker, noninterference guarantees that all information observable by the attacker is independent of confidential information. To make the connection between the host policy $\Gamma$, the database policy $\Sigma$ and the type system explicit we write $\Gamma, \Sigma \vdash e : t$ even though $\Sigma$ was kept implicit in the typing rules.
Theorem 1 (Soundness). If $x : t, \Sigma \vdash e : t'$, then $NI(e, e)_{t, \Sigma, t'}$.

Proof sketch. The theorem is proved by adapting the proof technique introduced by Pottier and Simonet [38] for an ML-like security-typed language. This is done by defining an extension of the language which allows reasoning about pairs of program configurations, and then showing that the type system for the extended language enjoys the subject reduction property. Then noninterference follows as a result of the subject reduction theorem. The proof can be found in the appendix.

The type system for the host language and the quoted language can be extended with two additional rules which take into account declassification through expressions from the set $D$. Intuitively, the rules allow to downgrade the security level of an expression if that expression is in the set of declassified expressions $D$ and the level $pc$ is upper bounded by the level of the declassified expression. The latter is used to enforce that no sensitive information is released implicitly through the declassification mechanism.

\[
\begin{align*}
&\text{DECL} \\
&\quad pc, \Gamma, M, D \vdash d : t \quad pc \sqsubseteq t 
\quad (d, t') \in D \\
&\quad pc, \Gamma, M \vdash d : t' \\
&\text{DECLQ} \\
&\quad H, \Delta, D \vdash d : t \quad pc \sqsubseteq t 
\quad (d, t') \in D \\
&\quad H, \Delta \vdash d : t'
\end{align*}
\]

Theorem 2 (Soundness under Declassification). If $x : t, \Sigma, D \vdash e : t'$, then $DNI(e, e)_{D, t, \Sigma, t'}$. 
\[
\begin{align*}
\textbf{Const} & \\
\Sigma(c) &= t \\
\text{pc}, \Gamma, M \vdash c : t^\ell \\
\textbf{Unit} & \\
\text{pc}, \Gamma, M \vdash () : \text{unit} \\
\textbf{Var} & \\
\quad x : t \in \Gamma & \\
\text{pc}, \Gamma, M \vdash x : t \\
\textbf{Nil} & \\
\text{pc}, \Gamma, M \vdash [] : (t \text{list})^\ell \\
\textbf{Loc} & \\
\quad l : t \in M & \\
\text{pc}, \Gamma, M \vdash l : t \\
\textbf{Fun} & \\
\text{pc}, \Gamma, x : t, M \vdash e : t' & \\
\text{pc}', \Gamma, M \vdash \text{fun}(x) \to e : (t \overset{pc}{\to} t') \\
\textbf{Rec} & \\
\text{pc}, \Gamma, x : t, M \vdash f : t \overset{pc}{\to} t', M \vdash e : t' & \\
\text{pc}', \Gamma, M \vdash \text{rec } f(x) \to e : t \overset{pc}{\to} t' \\
\textbf{Lift} & \\
\text{pc}, \Gamma, M \vdash e : t & \\
\text{pc}, \Gamma, M \vdash \text{lift } e : \text{Expr}(t) \\
\textbf{Exists} & \\
\text{pc}, \Gamma, M \vdash e : (t \text{list})^\ell & \\
\text{pc}, \Gamma, M \vdash \text{exists } e : \text{bool}^\ell \\
\textbf{Op} & \\
\quad \Sigma(op) = \bar{t} \to t & \\
\text{pc}, \Gamma, M \vdash \text{op}(\overline{e}) : t_i \downarrow t_i \\
\textbf{Yield} & \\
\text{pc}, \Gamma, M \vdash e : t & \\
\text{pc}, \Gamma, M \vdash \text{yield } e : (t \text{list})^\ell \\
\textbf{Apply} & \\
\text{pc}, \Gamma, M \vdash e_1 : t & \\
\text{pc}, \Gamma, M \vdash e_2 : t & \\
\text{pc} \sqsubseteq \text{pc}' & \\
\text{pc}, \Gamma, M \vdash e_1 \ e_2 : t' \\
\textbf{Pair} & \\
\text{pc}, \Gamma, M \vdash e_1 : t_1 & \\
\text{pc}, \Gamma, M \vdash e_2 : t_2 & \\
\text{pc}, \Gamma, M \vdash (e_1, e_2) : t_1 \times t_2 \\
\textbf{Fst} & \\
\text{pc}, \Gamma, M \vdash \text{fst } e : t_1 \\
\textbf{Snd} & \\
\text{pc}, \Gamma, M \vdash \text{snd } e : t_2 \\
\textbf{Record} & \\
\text{pc}, \Gamma, M \vdash \{f = e\} : \{f : t\} \\
\textbf{Project} & \\
\text{pc}, \Gamma, M \vdash \{f : t\} \\
\text{pc}, \Gamma, M \vdash e.f_i : t_i 
\end{align*}
\]
3.2. Framework

\[
\text{UNION} \quad pc, \Gamma, M \vdash e : (t \text{ list})^\ell \quad pc, \Gamma, M \vdash e' : (t \text{ list})^{\ell'} \quad pc, \Gamma, M \vdash e' @ e : (t \text{ list})^{\ell \sqcup \ell'}
\]

\[
\text{FOR} \quad pc, \Gamma, M \vdash e : (t \text{ list})^\ell \quad pc, \Gamma, x : t, M \vdash e' : (t' \text{ list})^{\ell'} \quad pc, \Gamma, M \vdash \text{for } x \text{ in } e \text{ do } e' : (t' \text{ list})^{\ell \sqcup \ell'}
\]

\[
\text{IF} \quad pc, \Gamma, M \vdash e : \text{bool}^\ell \quad pc, \Gamma, M \vdash e' : (t \text{ list})^{\ell'} \quad pc, \Gamma, M \vdash \text{if } e \text{ then } e' : (t \text{ list})^{\ell \sqcup \ell'}
\]

\[
\text{IF} \quad pc, \Gamma, M \vdash e : \text{bool}^\ell \quad pc \sqcup \ell, \Gamma, M \vdash e_i : t \quad \ell \sqsubseteq t \quad i \in \{1, 2\} \quad pc, \Gamma, M \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t
\]

\[
\text{DEREF} \quad pc, \Gamma, M \vdash e : t \text{ ref}^\ell \quad \ell \sqsubseteq t \quad pc, \Gamma, M \vdash e : t
\]

\[
\text{QUOTE} \quad pc, \Gamma, M, \cdot \vdash e : t \quad pc, \Gamma, M \vdash \langle@ e @\rangle : \text{Expr}(t)
\]

\[
\text{SUB} \quad t \sqsubseteq t' \quad pc, \Gamma, M \vdash e : t \quad pc, \Gamma, M \vdash e : t'
\]

\[
\text{RUN} \quad pc, \Gamma, M \vdash e : \text{Expr}(t) \quad pc, \Gamma, M \vdash \text{run } e : t
\]

\[
\text{REF} \quad pc, \Gamma, M \vdash e : t \quad pc \sqsubseteq t \quad pc, \Gamma, M \vdash \text{ref } e : t \text{ ref}^{pc}
\]

\[
\text{ASSN} \quad pc, \Gamma, M \vdash e_1 : t \text{ ref}^\ell \quad pc, \Gamma, M \vdash e_2 : t \quad pc \sqcup \ell \sqsubseteq t \quad pc, \Gamma, M \vdash e_1 := e_2 : \text{unit}
\]

Figure 3.3: Type system for host language
Figure 3.4: Typing rules for quoted language

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstQ</td>
<td>[ \Sigma(c) = t ]</td>
</tr>
<tr>
<td>H, \Delta \vdash c : t^\ell</td>
<td></td>
</tr>
<tr>
<td>FunQ</td>
<td>[ H, \Delta, x : t \vdash e : t' ]</td>
</tr>
<tr>
<td>H, \Delta \vdash \text{fun}(x) \to e : t \to t'</td>
<td></td>
</tr>
<tr>
<td>VarQ</td>
<td>[ x : t \in \Delta ]</td>
</tr>
<tr>
<td>H, \Delta \vdash x : t</td>
<td></td>
</tr>
<tr>
<td>ApplyQ</td>
<td>[ H, \Delta \vdash e_1 : t \to t' \quad H, \Delta \vdash e_2 : t ]</td>
</tr>
<tr>
<td>H, \Delta \vdash e_1 , e_2 : t'</td>
<td></td>
</tr>
<tr>
<td>OpQ</td>
<td>[ \Sigma(op) = t \to t ]</td>
</tr>
<tr>
<td>H, \Delta \vdash op(M) : t^\ell</td>
<td></td>
</tr>
<tr>
<td>Antiquote</td>
<td>[ H \vdash e : \text{Expr}(t) ]</td>
</tr>
<tr>
<td>H, \Delta \vdash \text{%}(e) : t</td>
<td></td>
</tr>
<tr>
<td>PairQ</td>
<td>[ H, \Delta \vdash e_1 : t_1 \quad H, \Delta \vdash e_2 : t_2 ]</td>
</tr>
<tr>
<td>H, \Delta \vdash (e_1, e_2) : t_1 \ast t_2</td>
<td></td>
</tr>
<tr>
<td>FstQ</td>
<td>[ H, \Delta \vdash \text{fst} , e : t_1 ]</td>
</tr>
<tr>
<td>SndQ</td>
<td>[ H, \Delta \vdash e : t_1 \ast t_2 ]</td>
</tr>
<tr>
<td>H, \Delta \vdash \text{snd} , e : t_2</td>
<td></td>
</tr>
<tr>
<td>RecordQ</td>
<td>[ H, \Delta \vdash M : t ]</td>
</tr>
<tr>
<td>H, \Delta \vdash { f = M } : { f : t }</td>
<td></td>
</tr>
<tr>
<td>ProjectQ</td>
<td>[ H, \Delta \vdash L : { f : t } ]</td>
</tr>
<tr>
<td>H, \Delta \vdash L.f_i : t_i</td>
<td></td>
</tr>
<tr>
<td>YieldQ</td>
<td>[ H, \Delta \vdash M : t ]</td>
</tr>
<tr>
<td>H, \Delta \vdash \text{yield} , M : (t , \text{list})^\ell</td>
<td></td>
</tr>
<tr>
<td>NilQ</td>
<td>[ H, \Delta \vdash [] : (t , \text{list})^\ell</td>
</tr>
<tr>
<td>ExistsQ</td>
<td>[ H, \Delta \vdash M : (t , \text{list})^\ell</td>
</tr>
<tr>
<td>H, \Delta \vdash \text{exists} , M : \text{bool}^\ell</td>
<td></td>
</tr>
<tr>
<td>IfQ</td>
<td>[ H, \Delta \vdash L : \text{bool}^\ell \quad H, \Delta \vdash M : (t , \text{list})^\ell</td>
</tr>
<tr>
<td>H, \Delta \vdash \text{if} , L , \text{then} , M : (t , \text{list})^{\ell \cup \ell'}</td>
<td></td>
</tr>
<tr>
<td>UnionQ</td>
<td>[ H, \Delta \vdash M : (t , \text{list})^\ell \quad H, \Delta \vdash N : (t , \text{list})^\ell</td>
</tr>
<tr>
<td>H, \Delta \vdash M , \text{@} , N : (t , \text{list})^{\ell \cup \ell'}</td>
<td></td>
</tr>
<tr>
<td>ForQ</td>
<td>[ H, \Delta \vdash M : (t , \text{list})^\ell \quad H, \Delta, x : t \vdash N : (t' , \text{list})^\ell</td>
</tr>
<tr>
<td>H, \Delta \vdash \text{for} , x , \text{in} , M , \text{do} , N : (t' , \text{list})^{\ell \cup \ell'}</td>
<td></td>
</tr>
<tr>
<td>SubQ</td>
<td>[ t \sqsubseteq t' ]</td>
</tr>
<tr>
<td>H, \Delta \vdash M : t</td>
<td></td>
</tr>
<tr>
<td>H, \Delta \vdash M : t'</td>
<td></td>
</tr>
<tr>
<td>DatabaseQ</td>
<td>[ \Sigma(db) = { f : t } ]</td>
</tr>
<tr>
<td>H, \Delta \vdash \text{database}(db) : { f : t }</td>
<td></td>
</tr>
</tbody>
</table>
3.3 JSLINQ

Figure 3.5 shows the architecture of JSLINQ. The input is an F# project consisting of the security policy and the application code. The right branch of the figure shows how a project is first compiled to a 3-tier application using the unmodified build process for web applications based on WebSharper. The code of the project is used to create a 3-tier application consisting of JavaScript created using WebSharper, .NET assemblies for server-side logic and SQL queries for the database, created using LINQ. Upon successful compilation, JSLINQ’s security type checker can be used on the F# project to determine if the application complies to the specified information-flow policy. How the resulting 3-tier application and the verification result are used depends on the use case of JSLINQ: one possibility is to discard non-compliant application builds and to deploy compliant applications into production. The remainder of the section discusses JSLINQ components in more detail.

WebSharper: WebSharper is a fully-featured and commercially supported framework for web application development in F#, providing powerful functional abstractions such as sitelets for document definition, formlets for data entry forms and flowlets for workflows [21]. Moreover, it offers abstractions for essential web concepts such as the DOM or JavaScript code. Importantly, these abstractions enjoy type safety properties, allowing to leverage the F# type system to build robust applications. One of WebSharper’s key features is the translation of F# functions into JavaScript code for execution in the browser. Server-side functions can be designated as remote procedure calls (RPC), and can be transparently

Figure 3.5: JSLINQ Architecture
called in client-side code, as in the example:

```fsharp
// Server-side function called by the client via AJAX.
[<Remote>]
let getText () = "JSLINQ"
// Client-side function translated to JS and HTML.
[<JavaScript>]
let Main () = Text (getText ())
```

WebSharper supports extensions of the client with third-party libraries, for example a map service. Third-party libraries usually consist of JavaScript code that is embedded into the page. Calls from the client-side F# code to the embedded third-party library are handled by wrappers that provide an F# interface to the JavaScript code. This approach requires full trust on the JavaScript code provided by the third party. However, JSLINQ can be used to type-check third-party libraries written in F#. This allows rewriting crucial third-party JavaScript libraries in F# to make them amenable to security analysis using JSLINQ.

**F# Project:** JSLINQ is designed to perform the verification step after successful compilation of the project. JSLINQ processes MSBuild projects and it is integrated with Microsoft Visual Studio. Code within a project is either part of the policy or part of the program. The policy controls information flows via security type signatures which are added to the definitions of functions and databases. The program implements the application and is subject to the security type check according to the policy. Since the policy is expressed within normal F# syntax, the use of JSLINQ does not interfere with the normal build process of the application and the use of standard tools.

**Policy:** The policy is specified by adding custom attributes with security type signatures to declarations. Signatures are represented as strings that follow the language in Section 3.2.1, and use variables for security levels in order to support polymorphism. If no security level is specified within a signature, the corresponding level variable is unconstrained. The following code fragment demonstrates how signatures are added to F# declarations:

```fsharp
[<SecT("_\^H")>]
let boolH = true
[<SecT("unit ->\^L _\^L")>]
```
let f () = 1

We divide a web-application policy into three types: a library policy, an RPC policy and a database policy. Each type deals with different tiers and the meaning of a security type signature depends on the tier in which it is located.

The policy for library functions is defined in a separate module, which is marked with a policy attribute. All library functions used by the program need to be wrapped in the policy, otherwise their use is not allowed. Since HTML and JavaScript abstractions of WebSharper are also library functions, the policy for client-side functionality is specified in this part. Each wrapper function has a mandatory security type signature that governs which security levels are used when the wrapper is called. The following snippet demonstrates a wrapper that uses WebSharper functions to generate a masked input field for passwords, labelled as high:

```ml
[<Policy>]
module Policy =
[<JavaScript>][<SecT("unit -> _^H")>]
let InputPW () = Input [Attr.Type "password"]
```

The policy for RPCs from the client to the server consists of attributes to the declarations of RPC functions within the program. We define the RPC policy and the program in the same file for sake of simplicity. However, JSLINQ allows a complete separation of policy and program into separate files, as we do for the other parts of the policy. Type signatures on RPC functions restrict the information flow from the client to the server (via function arguments) and from the server to the client (via return values). The following fragment demonstrates flows in both directions:

```ml
[<Remote>][<SecT("unit -> _^L")>]
let untrustedClient () = true

[<Remote>][<SecT("_^L -> ^L unit")>]
let untrustedServer (x:bool) = ()
```

The database policy is defined by adding security type signatures to an attribute-based mapping for LINQ [3]. Security type signatures are added to table and column definitions as shown in the following example:

```ml
[<Table>][<SecT("^L")>] // Public table length
```
Security Type Checker: The design of JSLINQ as a verification step after compilation allows us to assume that the code has correct syntax, data types and satisfied dependencies, hence the implementation can only focus on the security type check. Note-worthy, we leave the F# type system untouched and maintain a completely separate security type system during the verification. We perform the security type checking in two steps, which we repeat for each top-level declaration found in the code: first we recursively traverse AST for the declaration to obtain set of constraints and a security type signature by means of the FParsec library [6]. The second step substitutes level variables with actual security levels by solving the constraint set. The resulting types and possibly remaining constraints are added to the environment before proceeding with the next declaration. JSLINQ uses the AST generated by the F# compiler, which is retrieved using the library FSharp Compiler Services [5]. We thus do not duplicate compiler features that are unrelated to the security type check and benefit from F#'s desugaring. This is a clear advantage over prototypes, e.g. SELINQ or SIF, that enhance existing type systems.

3.4 Case Studies

We have used JSLINQ to implement several case studies as F# projects. In this section we first describe the general design of the policy language and then remark on the policy requirements for the case studies that we have implemented.

3.4.1 Library Policy

The largest part of the library policy are the signatures for the DOM and JavaScript abstractions. The documents shown in the browser are constructed using these abstractions at runtime. For simplification, we consider the HTML elements as trusted sinks. The rationale behind this is that the user has full access to the
data once it has arrived in the browser, independently of that data being displayed or not. However, this assumption does not hold for the full WebSharper API, as it would allow to write and read the elements in the DOM tree in various ways. Therefore, the policy only permits basic operations on the DOM. An important exception from our trusted sink assumption are HTML elements which load external resources, such as images and IFrames. These elements can be used to leak data either directly within the source attribute or indirectly via externally observable HTTP requests. Therefore, we annotate the creation of the source attribute with low security level, both for the URL argument and the side-effects.

### 3.4.2 Scenario Discussion

We now comment on different aspects of the policy and provide examples for vulnerabilities captured by JSLINQ.

**Password Meter** We have included the password meter to demonstrate a policy with full client isolation, where the password is not allowed to leave the browser. The policy declares password fields as sensitive sources. Leaks to third parties and to the application server are prevented by assigning low levels to the source attribute and to the arguments and side-effects of RPC functions, respectively. The scenario assumes that the server is untrusted, as it should not receive the password. A problem with this view is that the JavaScript code executed by the client is usually delivered by an untrusted server. This means that the integrity of the client-side code after the security type check is not guaranteed. Such changes are not subject to the security policy and can thus be abused to leak confidential data. Therefore we have to put trust in the integrity of the code delivered by the application server, which we summarize as *partial trust*. Alternatively, remote attestation methods such as code or certificate signatures can ensure code integrity. The following snippets show a secure password check and two leaks via the source attribute that are handled correctly by JSLINQ. The scenario consists of 53 F# and 6215 generated JS LOCs.

```fsharp
let content = // Allowed: Secret only in browser.
    if (containsLetters password)
        then Text "Passed" else Text "Failed"
```
let content' = // Blocked: Leak via source attribute.
Image [Src ("http://example.com/img.png?"+password)]
// Blocked: Leak via side-effects.
let content'' = Src (if secret == "jSL!Nq42"
    then "http://example.com/true.jpg"
    else "http://example.com/false.jpg")

Location-Based Service This scenario demonstrates declassification of a client-side secret, in this case the user’s position. Third parties and the application server can only receive declassified (obfuscated) coordinates. We define declassification as a function that adds a random offset to the position. The function is applied to the confidential latitude and longitude values. The real coordinates are isolated in the browser in the same way as for the password meter. We provide two variants of the location-based service to showcase two different attacker models. The first example embeds a map via an IFrame, where the position is an argument to the source attribute of the IFrame. The following snippet shows how the use of declassified coordinates is permitted, while real coordinates are blocked:

let iframeSrc = Src // Allowed: Obfuscated coordinate.
  "https://maps.example.com/?q=
  (string (randomize Lat)) + "," +
  (string (randomize Lon))

let iframeSrc' = Src // Blocked: Exact coordinate.
  "https://maps.example.com/?q=
  (string Lat) + "," + (string Lon)

The second example includes a third-party library called via F#. We use the Google Maps extension for WebSharper and wrap the initialization and panning of the map within the policy, both having low side-effects and low values. Since the extension wraps the original JavaScript code, we have to fully trust the F#-to-JavaScript extension and JavaScript code implementing the WebSharper APIs. The scenario consists of 76 F# and 6279 generated JS LOCs.

Movie Rental This scenario demonstrates the use of security policies on databases. The database consists of a list of items (e.g. movies) subject to events (e.g. movie rentals) happening at a
certain location and time. The location of an event is confidential, while all other information is public. The database policy assigns to the latitude and longitude high-security levels. Leaks to the client are prevented by labelling the return values of RPC functions as public. The following LINQ query joins rentals with movies and returns a list of movie titles. Movie titles are input to an RPC function which is only allowed to return public values. As a result the first `yield` statement is allowed to return the movie titles. If instead we use the second `yield` statement, JSLINQ rejects the program.

```fsharp
let events = query {
    for e in db.Event do
    for i in db.Item do
        if e.ItemId = i.Id then
            (* Allowed *) yield i.Name
            (* Blocked *) yield (string e.Lat)
}
```

Moreover, we allow the user to retrieve a ranking of popular movies within an area. The implementation contains a pre-defined set of areas which are addressed using indexes. The user can only specify the index for an area of interest. The application server filters the list of movie rentals based on the coordinate values. JSLINQ will infer a high-security level for the length of the resulting list, as it depends on the coordinate values. Our policy allows that geographic information about rentals is disclosed on the granularity of fixed-size areas, therefore we can directly declassify the length of the list. The scenario consists of 87 F# and 6231 generated JS LOCs.

**Friend Finder App** In this scenario we consider a completely untrusted application server. The client obtains the code from a trusted source. We use the Apache Cordova framework [2] to package the client-side functionality as an app that can be distributed via a trusted channel. Cordova also provides access to the address book of the device. The app can access the address book only via a function defined in the policy, which assigns a high-security level to the contact details. The policy allows declassification by means of a hash function on strings. Leakage of plain contact details to the untrusted server is prevented by assigning a low security level to the arguments and side-effects of RPC functions. The following snippet illustrates a secure and an insecure RPC call:
// Allowed: Look-up of hashed phone number
let rpcResult = remoteLookup (Hash phoneNumber)

// Blocked: Look-up of plain phone number
let rpcResult' = remoteLookup phoneNumber

The scenario has 62 F# and 9966 generated JS LOCs.

**Battleship** We implement a simplified version of the classical Battleship game [29, 39]. The client uses the browser to play against the server and the goal of each player is to hide the exact position of their ships on a grid. Both sides trust each other to correctly follow the rules of the game, so we are only concerned about confidentiality. A desirable IFC policy for this game is to mark the values indicating individual ship positions as confidential and all parameters and return values of RPC functions as public, so that confidential information is not allowed to pass the barrier between the browser and the server. This allows us to re-use the same security policy on both sides, as shown in Figure 3.6. The game rules require declassification, since the response to a shot requires disclosure of one bit of information ("hit" or "miss") to the other player per round. On each side we have to perform declassification twice: firstly for the hit/miss response to a shot, as it directly depends on the presence of a ship at that location, and secondly for indicating to the opponent if a player is defeated, which requires to test all occupied cells. The latter can be done locally, but for implementation reasons players report their own defeat to the opponent. The following example shows this for the client-side:

```fsharp
let serverShotResult = {
    shot = response.shot;
    hit = DeclassifyBool !serverTarget.occupied;
    defeated = DeclassifyBool clientDefeated }
```

The scenario has 255 F# and 6348 generated JS LOCs.

### 3.4.3 Case Study Results

Table 3.1 summarizes our case studies. The different combinations of client, third party and server trust illustrate the attacker models handled by JSLINQ. The initial effort of defining the API policy annotations comes with the benefit of minor burden on application programmer side. The policy for JSLINQ requires only very few annotations within the application code. As reported above,
Figure 3.6: Simplified IFC policy for Battleship

Table 3.1: Overview of implemented scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Trust</th>
<th># of Annotations</th>
<th># of Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password Meter</td>
<td>Yes</td>
<td>No</td>
<td>Partial</td>
</tr>
<tr>
<td>POI IFrame</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>POI Embedded</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Movie Rental</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Friend Finder</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Battleship</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

the LOCs for F# and JavaScript refer to the application (excluding comments and blank lines) and wrappers in the policy. The difference between the number of lines in F# code and resulting JavaScript shows WebSharper and its libraries at work. This allows the programmer to focus on the application logic and its security-critical parts (subject to security type check in JSLINQ) while standard boilerplate code is automatically generated by the framework. Real-world applications contain considerably more code to offer a better user experience. We omit the verification time, as execution time mostly consists of the compilation required to retrieve the AST. As the security type check is based on a simple constraint solver, we expect it to scale well to larger programs.

3.5 Related Work

Securing web applications with IFC has been the subject of a large array of research studies. Here we contrast our approach with
Table 3.2: Comparison of web application frameworks

<table>
<thead>
<tr>
<th>Tool</th>
<th>Client</th>
<th>Server</th>
<th>DB</th>
<th>3rd Party</th>
<th>Dec</th>
<th>Sound Core</th>
<th>Enforcement</th>
<th>Language</th>
<th>P#C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIF/SWIFT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>TS</td>
<td>Java, HTML</td>
<td>✓</td>
</tr>
<tr>
<td>WebSSARI</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>TS</td>
<td>PHP, SQL</td>
<td>✓</td>
</tr>
<tr>
<td>IFDB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Dynamic</td>
<td>PHP, SQL</td>
<td>✓</td>
</tr>
<tr>
<td>SELINKS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>TS</td>
<td>Links</td>
<td>✓</td>
</tr>
<tr>
<td>UR/WEB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>ATP</td>
<td>UR</td>
<td>✓</td>
</tr>
<tr>
<td>SELINQ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>TS</td>
<td>F#</td>
<td>✓</td>
</tr>
<tr>
<td>JSFLOW</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Dynamic</td>
<td>JavaScript</td>
<td>✓</td>
</tr>
<tr>
<td>JSLINQ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>TS</td>
<td>F#</td>
<td>✓</td>
</tr>
</tbody>
</table>

closely related works on IFC for web security.

**Information Flow Security.** Much research on formal models for end-to-end security guarantees has followed Goguen and Meseguer’s seminal work on noninterference [20]. Heintze and Riecke [24] introduce the SLam calculus to enforce noninterference for a functional language with higher-order features and present a soundness proof for a functional fragment of that language. Potter and Simonet [30] introduce a security type system for a core of ML with references and higher-order features and implement type checking for the FlowCaml tool [38]. Our framework extends the soundness proof technique from [30] with support for higher-order types, quotations and antiquotations, and declassification. A plethora of static, dynamic and hybrid analysis have been proposed to enforce noninterference-like policies [34]. Our work uses static analysis by means a security type system.

**Web Application Security.** Common security mechanisms proposed for web applications, including IFC, only secure components in isolation. Database systems such as MySQL provide access controls at the level of tables and columns, which are decoupled from the applications. Similarly, web browsers [23, 13] and application servers [34, 22] leverage dynamic and static techniques to enforce policies in isolation. None of these approaches can express security policies that regulate information flows across component boundaries as we do in this paper. Many existing web application frameworks augment the capabilities of a specific language with homogeneous meta-programming to ease the construction of Internet applications. WebSharper, Rails, GWT and many others are used in industry to develop complex web and mobile applications. For instance, GWT is used by many products at Google, including Flights, Hotel Finder, Offers and Wallet. While there is some framework support as prepared statements and custom sanitizers,
the burden of securing code is largely placed on the developer. JS-
LINQ provides a smooth integration of security requirements in
the development process, which allows F# programmers to check
whether their code, or the code developed by external contractors,
complies with desired security policies.

A few existing works aim at bridging IFC for multi-tier web
applications. Chong et al. implement SIF [17] and SWIFT [16]
as extensions of the JIF compiler [29] to enforce information flow
policies for web applications written in Java. Web applications
are checked against these policies by a combination of static and
runtime enforcements. The ability to enforce fine-grained policies
in the decentralized label model [28] is an attractive feature. At
the same time, SIF and SWIFT interweave security annotations
with program code and do not provide support for databases. JS-
LINQ addresses soundness formally and provides integration for
third-party libraries. Huang et al. [25] propose WebSSARI, a tool
that combines static analysis with runtime checks to detect vul-
nerabilities in PHP applications that interact with SQL databases.
WebSSARI is very effective at discovering security vulnerabilities,
although no support for client-side applications is provided and
soundness is only addressed informally. Schultz and Liskov [37]
propose IFDB, a database management system with decentralized
IFC. IFDB is implemented by modifying PostgreSQL as well as
the application environments in PHP and Python. Their Query
by Label model provides abstractions for dealing with expressive
information flow policies in relational databases, including decen-
tralization and declassification. IFDB supports policies for server
and database tiers and does not provide language integration for
database queries. Corcoran et al. [18] present SELINKS which
builds on the Links programming language. Links is a strongly-
typed functional language for multi-tier web applications and it
supports higher-order queries. SELINKS implements an expressive
type system which allows to define a variety of policies, including
dynamic IFC, provenance, and general access control. JSLINQ
only requires the programmer write code in a mainstream lan-
guage such as F# and express policies in a less sophisticated, but
standard type system. Chlipala introduces UrFlow [15], which im-
plements a static information flow analysis as part of the Ur/Web
domain-specific language for development of web applications. Ur-
Flow allows to express policies as SQL queries leveraging the users’ runtime knowledge. The enforcement is done by symbolic execution over a model of the web application. UrFlow shares similar aspects with SELINKS and scalability depends on capabilities of the underlying theorem prover. While JSLINQ separates security checking from type checking, it can be extended with techniques from [43] to cope with dynamic security policies. Hedin et al. [23] present JSFlow, a security-enhanced JavaScript interpreter for fine-grained tracking of information flow. The interpreter enables deployment as a browser extension providing dynamic IFC on the client-side including third-party scripts. JSFlow only applies to applications written in JavaScript.

**Secure Compilation** JSLINQ relies on the WebSharper compiler to translate F# code to JavaScript code deployed in the web browser, leaving out a formal investigation of the translation correctness. Fournet et al. [19] show full abstraction for a compiler which translates an ML-like language with higher-order functions and references to JavaScript. Their language is similar to F#, hence the same ideas can be used to show full abstraction for the JSLINQ compiler. Baltopoulos and Gordon [12] study secure compilation by augmenting the Links compiler with encryption and authentication for data stored on the client-side.

**Tools** Table 3.2 provides a comparison of existing web application frameworks with support for IFC. We classify each tool depending on whether they allow for IFC on the client, server, databases (DB) or third-party libraries. We also compare against support for declassification policies (Dec), soundness of a core calculus, type of enforcement mechanism (a type system (TS), a dynamic monitor or an automated theorem prover (ATP)), programming languages used and separation between code and policy (P#C). The comparison shows that JSLINQ enjoys many desirable properties.

### 3.6 Conclusion

We have presented a framework for end-to-end security, by leveraging IFC for a functional language with mutable store and language-integrated queries. The framework puts homogeneous meta-programming to work by developing a security type system that tracks in-
formation flows through the client, server, and underlying database. We have implemented JSLINQ and shown through different case studies that it is practical. JSLINQ can be used by organizations to build high-assurance applications. It can automatically verify the information flows within code written by internal developers or external contractors against the security policy. This helps to improve code quality and to demonstrate compliance with information security regulations, for instance when sensitive information like trade secrets or personal data is being processed. As future work, we plan to add to JSLINQ support for dynamic policies and finer-grained third-party libraries from F# and ensure their secure compilation to JavaScript.

Acknowledgments This work was funded by the European Community under the ProSecuToR project and the Swedish research agencies SSF and VR.

Bibliography


3.A. Appendix

3.A.1 Operational Semantics

Following Pottier and Simonet [30], the noninterference proof is reduced to a subject reduction proof for an extended language and an extended type system. Noninterference requires to reason about executions of two terms $e_1$ and $e_2$, and show they are related with respect to observations at security level $\ell$. The extended language provides a syntactic way to reason about execution pairs by introducing a bracket construct $\langle e_1 \mid e_2 \rangle$, which represents an execution
Figure 3.9: Evaluation rules for host language

pair as a single term. We refer to a term within brackets as a binary term and to a term without brackets and a unary term. Given a term $e$ with free variables $\overline{x}$ and two related values $\overline{v}_1$ and $\overline{v}_2$, the execution of $e[\overline{v}_1/\overline{x}]$ and $e[\overline{v}_2/\overline{x}]$ can be incorporated into a term $e[⟨\overline{v}_1 | \overline{v}_2⟩/\overline{x}]$ in the extended language. We use this to show
that two terms only differ on the confidential part if they can be
encoded by a well-typed term in the extended language. Therefore,
proving the noninterference of the original language is reduced to
proving the subject reduction theorem of the extended language.

We extend the language syntax with the bracket construct both
for terms and values. A new value \texttt{void} is used to represent cases
where the memory is unbound for one of the terms, and it is com-
patible with any type.
\( e ::= \ldots \mid \langle e \mid e \rangle \)

\( v ::= \ldots \mid \text{void} \mid \langle v \mid v \rangle \)

The subterms of the bracket construct are either \text{void} or unary terms, and brackets can not be nested. \textit{Projection} functions \([\bullet]_i\), with \(i \in \{1, 2\}\), are used to establish the correspondence between binary terms and unary terms. Given a term \(e\), the function \([e]_i = e_i\) if \(e = \langle e_1 \mid e_2 \rangle\), otherwise it represents identity.

The presence of mutable storage requires to keep track of binary values shared between stores. Since memories may have distinct domains, the bindings of the form \(l \mapsto (v|\text{void})\) and \(l \mapsto (v|\text{void})\) represent cases where location \(l\) is bound within only one of the two memories. The projection function is extended to memories as expected. Given a configuration \((e, \mu)\), then \([\mu]_i\) maps location \(l\) to \([\mu(l)]_i\) iff the latter is defined and is not \text{void}. Moreover, the projection \([\langle e, \mu \rangle]_i\) is defined as \(([e]_i, [\mu]_i)\).

The operational semantics of binary terms can be expressed in terms of operational semantics of respective unary terms, as defined in the previous sections. An evaluation step of a bracket expression \(\langle e_1 \mid e_2 \rangle\) is an evaluation step of either \(e_1\) or \(e_2\) which can only access the corresponding projection of the memory. A configuration has an index \(i \in \{\bullet, 1, 2\}\) that indicates whether the term to be evaluated is a subterm of a binary term, and if so which branch of a bracket the term belongs to. For example, the configuration \([\langle e, \mu \rangle]_1\), or simply \((e, \mu)_1\), means that \(e\) belongs to the first branch of a bracket, and it can only access the first projection of \(\mu\). Moreover \((e, \mu)_\bullet\), or simply \((e, \mu)\), denotes a unary configuration.

\textbf{Operational Semantics} \hspace{1em} The operational semantics rules of the extended language are given in Fig. 3.12. The semantics of unary reductions defined earlier (Fig. 3.9) applies to projections of binary terms, with a few twists regarding memory operations. The new reduction rules allow to manipulate bracket constructs, i.e., keep track of the information flows, and they do not have any computational effect on the respective projections. The purpose of \textit{lifting} rules is to prevent the binary terms from blocking the execution. This is achieved by duplicating the shared subterm in a bracket and thus allowing the execution to proceed independently within each branch. The memory rules are modified to access the store in a context-dependent manner. In fact, the memory projection of
index \(i\) forces reductions inside brackets to only affect the \(i\)-th projection of the store. The bracket construct is just a syntactic sugar to encode executions pairs and it does not have any computational effect, as shown by the following lemmas:

**Lemma 1** (Soundness). If \((e, \mu) \rightarrow (e', \mu')\), then \([((e, \mu)]_i \rightarrow [((e', \mu')]_i\), where \(i \in \{1, 2\}\).

*Proof.* The lemma can be shown by inspection of the evaluation rules. \(\square\)

**Lemma 2** (Completeness). Suppose \([((e, \mu)]_i \rightarrow^* (v_i, \mu'_i)\), where \(i \in \{1, 2\}\). Then, there exists \((v, \mu')\) such that \((e, \mu) \rightarrow^* (v, \mu')\).

*Proof.* We show that \((e, \mu)\) does not admit an infinite evaluation sequence. First, infinite evaluations can not arise from lifting rules since these rules only move the bracket towards the term’s root, which by definition is finite. Furthermore, lifting rules have no computational effects, hence both projections of a configuration are left unchanged. As a result, an infinite evaluation sequence only arises whenever one of the projections \([((e, \mu)]_i\) admits such an infinite sequence. But this would contradict the assumption of the lemma, since the semantics is deterministic. On the other hand, configurations might get stuck and not produce a value. Again, we can show that \((e, \mu)\) gets stuck only if at least one of the projections \([((e, \mu)]_i\) gets stuck, which contradicts the assumptions of the lemma. \(\square\)

The completeness lemma shows that if both projections of a term can be reduced to a successful configuration, then so can the term itself. This means that we have provided enough lifting rules to allow reducing all meaningful binary terms.

**Security Type System** The security type system is extended with two typing rules to handle the bracket construct and the void values. Rule \textsc{Bracket} guarantees that binary terms are only typed in high security contexts. This reflects the intuition that binary terms encode branching under high conditions.
Lifting rules

\[
\begin{align*}
(\text{ref } v, \mu)_i & \rightarrow (l, \mu[l \mapsto \text{new}_i v])_i \\
l \not\in \text{dom}(\mu) & \\
(!l, \mu)_i & \rightarrow (\text{read}_i \mu(l), \mu)_i, \ l \in \text{dom}(\mu) \\
(l := v, \mu)_i & \rightarrow \\
(((), \mu[l \mapsto \text{update}_i \mu(l) v])_i, \ l \in \text{dom}(\mu) & \\
(op(\langle v_1 | v_2 \rangle, v), \mu) & \rightarrow (op(\langle v_1 v | v_2 v \rangle), \mu) \\
(\langle v_1 | v_2 \rangle v, \mu) & \rightarrow ((\langle v_1[v]_1 | v_2[v]_2 \rangle), \mu) \\
(!\langle l_1 | l_2 \rangle, \mu) & \rightarrow ((\langle !l_1 | !l_2 \rangle), \mu) \\
(\langle l_1 | l_2 \rangle := v, \mu) & \rightarrow ((\langle l_1 := [v]_1 | l_2 := [v]_2 \rangle, \mu) \\
(\langle v_1 | v_2 \rangle \text{ then } e_1 \text{ else } e_2, \mu) & \rightarrow \\
((\langle v_1 \text{ then } [e_1]_1 \text{ else } [e_2]_1 | \text{ if } v_2 \text{ then } [e_1]_2 \text{ else } [e_2]_2 \rangle, \mu) & \\
(\langle v_1 | v_2 \rangle \text{ then } e, \mu) & \rightarrow \\
((\langle \text{if } v_1 \text{ then } e | \text{ if } v_2 \text{ then } e \rangle), \mu) & \\
(lift \langle v_1 | v_2 \rangle, \mu) & \rightarrow ((\langle \text{lift } v_1 | \text{ lift } v_2 \rangle, \mu) \\
\end{align*}
\]

\[
\begin{align*}
(e_1, \mu)_i & \rightarrow (e'_i, \mu')_i \\
\{i, j\} & = \{1, 2\}
\end{align*}
\]

Auxiliary functions

\[
\begin{align*}
\text{new} \cdot v & = v & \text{update} \cdot v v' & = v' & \text{read} \cdot v & = v \\
\text{new}_1 v & = \langle v | \text{void} \rangle & \text{update}_1 v v' & = \langle v' | [v]_2 \rangle & \text{read}_1 v & = [v]_1 \\
\text{new}_2 v & = \langle \text{void} | v \rangle & \text{update}_2 v v' & = ([v]_1 | v') & \text{read}_2 v & = [v]_2
\end{align*}
\]

Figure 3.12: Evaluation rules for extended host language
The following lemmas are needed to prove the subject reduction theorem.

**Lemma 3** (Projection). If $pc, \Gamma, M \vdash e : t$ then $pc, \Gamma, M \vdash [e]_i : t$ for $i \in \{1, 2\}$. Similarly, if $H, \Delta \vdash e : t$ then $H, \Delta \vdash [e]_i : t$.

*Proof.* The lemma is proved by induction on derivation of the judgement. If $e$ is not a bracket, the lemma follows trivially. Otherwise, suppose $e = \langle e_1 \mid e_2 \rangle$. By the premisses of the bracket rule $H, \Gamma, M \vdash [e]_i : t$ and since $pc \subseteq H$, it follows that $pc, \Gamma, M \vdash [e]_i : t$. The proof for quoted judgements is similar. ∎

**Lemma 4** (Store Access). Let $i \in \{\bullet, 1, 2\}$ and suppose $pc, \Gamma, M \vdash v : t$ and $pc, \Gamma, M \vdash v' : t$. Moreover, if $i \in \{1, 2\}$ then $H \subseteq t$.

Then $pc, \Gamma, M \vdash \text{read}_i v : t$, $pc, \Gamma, M \vdash \text{new}_i v : t$ and $pc, \Gamma, M \vdash \text{update}_i v v' : t$.

*Proof.* The rule follows by the definition of the auxiliary functions for the memory (Fig. 3.12), the projection lemma 3 and the typing rules for bracket and void constructs. We show that $pc, \Gamma, M \vdash \text{new}_i v : t$ follows from $pc, \Gamma, M \vdash v : t$. By definition of $\text{new}_i v$ we have three cases: (a) if $i = \bullet$, then $\text{new}_\bullet v = v$, hence the lemma follows by assumption, (b) if $i = 1$, then $\text{new}_1 v = \langle v \mid \text{void} \rangle$. By the typing rule Bracket, the projection lemma 3 and the rule $\text{VOID}$ the claim follows immediately, (c) Symmetric to (b). ∎

**Lemma 5** (Substitution). Let $M \vdash v : t$ and $pc, \Gamma[x \mapsto t], M \vdash e : t'$. Then $pc, \Gamma, M \vdash e[x \mapsto v] : t'$.

*Proof.* The lemma is proved by induction on the derivation of the judgement $pc, \Gamma[x \mapsto t], M \vdash e : t'$. We show a few cases below.

Case $\text{VAR}$, $e = y$: If $y = x$, then since $e$ is well typed, $y$ occurs in the typing context and $y : t$ and $t = t'$. Moreover, $e[x \mapsto v] = v$ and
\(v : t'.\) Otherwise, if \(y \neq x\) then \(e[x \mapsto v] = e\) and by assumption \(e : t'.\)

Case \textsc{fun}, \(e = \text{fun}(y) \rightarrow e':\) By assumption, \(pc, \Gamma[x \mapsto t], M \vdash \text{fun}(y) \rightarrow e' : t_1 \overset{pc'}{\rightarrow} t_2,\) where \(t' = t_1 \overset{pc'}{\rightarrow} t_2\) and \(v : t.\) By the premise of rule \textsc{fun} \(pc', \Gamma[x \mapsto t][y \mapsto t'], M \vdash e' : t_2.\) By induction hypothesis \(pc', \Gamma, M \vdash e'[x \mapsto v] : t_2,\) hence the lemma follows by the premise of \textsc{fun}.

Case \textsc{if}, \(e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3:\) By assumption \(pc, \Gamma[x \mapsto t], M \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t'\) and \(v : t.\) By induction hypothesis we have \(pc, \Gamma, M \vdash e_i[x \mapsto v] : t_i'\) and by the premises of the rule \textsc{if}, the claim follows.

Case \textsc{bracket}, \(e = \langle e_1 | e_2 \rangle:\) By assumption, \(pc, \Gamma[x \mapsto t], M \vdash \langle e_1 | e_2 \rangle : t'\) and \(v : t.\) By induction hypothesis we have \(\mathcal{H}, \Gamma, M \vdash e_i[x \mapsto [v]_i] : t'\), hence the lemma follows by the premises of the rule \textsc{bracket} and the projection lemma 3.

\[\square\]

**Theorem 3** (Subject Reduction). Let \(pc, M \vdash e : t, M \vdash \mu\) and \((e, \mu)_i \rightarrow (e', \mu')_i\) for \(i \in \{\bullet, 1, 2\}\). Moreover, \(pc = \mathcal{H}\) if \(i \in \{1, 2\}\). Then there exists \(M'\) extending \(M\), such that \(pc, M' \vdash e' : t\) and \(M' \vdash \mu'.\)

**Proof.** The theorem is shown by induction on the derivation of evaluation \((e, \mu)_i \rightarrow (e', \mu')_i\). If the derivation of \(pc, M \vdash e : t\) uses the rule \textsc{sub}, then there is a \(t' \subseteq t\), such that \(pc, M \vdash e : t'\) does not end with an instance of \textsc{sub}. Hence, we can assume this is the case from now on without losing generality. Therefore, the derivation must end with a syntax directed rule which match the term \(e\).

Case \textsc{op}, \(e = op(\overline{v})\): We have \(\Sigma(op) = \overline{t} \rightarrow t'\) and \(e : t'^{\ell'}\), and \(pc, M \vdash e : t'^{\ell'}\) with \(\ell' = \bigsqcup \ell_i\). We assume that all built-in operators preserve the type, i.e. \(\forall op, \overline{v} : \overline{t} \Rightarrow \delta(op, \overline{v}) : t'.\) Then by induction hypothesis, the typing rule \textsc{op} and the assumption \(M \vdash \mu\), we have that \(pc, M' \vdash \delta(op, \overline{v}) : t'\) with \(M = M'\).

Case \textsc{fun}, \(e = (\text{fun}(x) \rightarrow e') v\): By rule \textsc{apply} we have \(pc, M \vdash \text{fun}(x) \rightarrow e' : t \overset{pc}{\rightarrow} t'\) and \(pc, M \vdash v : t.\) By rule \textsc{fun} we have that \(pc', [x \mapsto t], M \vdash e' : t'.\) We can then apply the substitution lemma 5 (modulo applications of rule \textsc{sub}) and prove that \(pc, M \vdash e'[x \mapsto v] : t'.\)
Case REC, $e = (\text{rec } f(x) \to e') v$: Similar to the previous case.

Case FST, $e = \text{fst} (v_1, v_2)$: By rule Fst we have $pc, M \vdash (v_1, v_2) : t_1 * t_2$ and by rule PAIR we have $pc, M \vdash v_1 : t_1$. Then the claim follows by induction hypothesis.

Case SND, $e = \text{snd} (v_1, v_2)$: Symmetric to the previous case.

Case PROJECT, $e = \{ f = v \}. f_i$: Follows immediately by rules PROJECT, RECORD and the induction hypothesis.

Case IF1, $e = \text{if } v_1 \text{ then } v_2$: By rule IF1 $t = (t' \text{ list}^{l,l'})$, $pc, M \vdash v_1 : \text{bool}^l$ and $pc, M \vdash v_2 : (t' \text{ list}^{l'})$. If $v_1 = \text{true}$ then $e' = v_2$, otherwise $e' = \[].$ By induction hypothesis the claim follows.

To prove noninterference for values of arbitrary types, as defined by the equivalence relations $\sim_t$ in Figure 3.3, we need to define an encoding of input values of a given type $t$. The encoding transforms a pair of values $v_1 \sim_t v_2$ into a single term $v$ using brackets whenever the component’s security label is typed as high. We then prove that the resulting value has type $t$ in the extended type system.

Definition 4 (Binary Encoding). Let $v_1$ and $v_2$ be two values such that $v_1 \sim_t v_2$. Then the encoding function Enc is recursively defined by the rules in Figure 3.13:

Lemma 6. If $v_1 \sim_t v_2$ and $v = \text{Enc}(v_1, v_2, t)$, then $\vdash v : t$.

Proof. Induction on $v$ and rules in Figure 3.13.

Then noninterference follows from the subject reduction theorem and the soundness and completeness of the extended language semantics. It is worth noting that the proof holds for multiple inputs $x : t$, since they can be encoded as records.

Proof of Theorem 1. If $x : t \vdash e : t'$, $e[x \mapsto v_1] \xrightarrow{\ast} \Omega_1 (v'_1, \mu'_1)$, $e[x \mapsto v_2] \xrightarrow{\ast} \Omega_2 (v'_2, \mu'_2)$, $\Omega_1 \sim \Omega_2$ and $v_1 \sim t v_2$, then $v'_1 \sim v' v'_2$.

Proof. Let $v = \text{Enc}(v_1, v_2, t)$. By Lemma 6, $\vdash v : t$. By substitution lemma 5, $\vdash e[x \mapsto v] : t'$. Then since $[e[x \mapsto v]]_i = e[x \mapsto v_i]$ and, by hypothesis, evaluates to $v'_i$ for $i \in \{1, 2\}$, we can use the completeness lemma 2 to show that $e[x \mapsto v] \xrightarrow{\ast} (v', \mu')$. By the subject reduction theorem 3 it follows that $\vdash v' : t'$. If $v'$ is a
\[
\begin{align*}
\frac{v_1 \sim c v_2}{\langle v_1 \mid v_2 \rangle} & \quad \frac{v_1 \sim c v_2}{v_1} & \quad \frac{(v_1, v_2) \sim t_1 \cdot t_2 (v'_1, v'_2)}{(v_1 \sim t_1, v_2 \sim t_2 v'_2)} \\
\{ f = v \} & \sim \{ f = w \} & \frac{[v] \sim (t \text{ list}) [w]}{\ell = L \Rightarrow [v] \sim \ell} \\
\frac{\ell = H \Rightarrow \langle [v] \mid [w] \rangle}{\text{fun}(x) \rightarrow e_1 \sim \ell \cdot t' \text{ fun}(x) \rightarrow e_2 \quad H \sqsubseteq \ell} & \quad \frac{\text{fun}(x) \rightarrow e_1 \mid \text{fun}(x) \rightarrow e_1}{t' \subseteq \ell} \\
\frac{\text{fun}(x) \rightarrow e_1 \sim \ell \cdot t' \text{ fun}(x) \rightarrow e_2}{\langle \text{rec } f(x) \rightarrow e_1 \mid \text{rec } f(x) \rightarrow e_1 \rangle} & \quad \frac{\text{rec } f(x) \rightarrow e_1 \sim \ell \cdot t' \text{ rec } f(x) \rightarrow e_2}{t' \subseteq \ell} \\
\text{rec } f(x) \rightarrow e_1 \sim \ell \cdot t' \text{ rec } f(x) \rightarrow e_2 & \quad v_1 \sim \text{Expr}(t) v_2
\end{align*}
\]

Figure 3.13: Value encoding

bracket, we are done. Otherwise, \(|v'_1|_1 = |v'_2|_2\). By the soundness theorem 1 and the determinism of the operational semantics, for \(i \in \{1, 2\}\), we have \(e[x \mapsto v_i] \xrightarrow{\ast} (v'_i, \mu'_i)_i\), hence \(|v'_1|_1 = |v'_2|_2\).

Proof of Theorem 2. If \(x : t, \Sigma, D \vdash e : t'\), \(e[x \mapsto v_1] \xrightarrow{\ast} (v_1', \mu_1')_1\), \(e[x \mapsto v_2] \xrightarrow{\ast} (v_2', \mu_2')_2\), \(\Omega_1 \sim \Sigma \Omega_2\), \(d_j[x \mapsto v_1] \sim \Sigma d_j[x \mapsto v_2]\) for \(d_j \in D\) and \(v'_1 \sim t \cdot v'_2\).

Proof. Similar to Theorem 1. The only difference is whenever rules \(\text{DECL}\) and \(\text{DECLQ}\) apply. In that case the claim follows by the assumptions of the theorem and subject reduction 3.