Prudent Design Principles for Information Flow Control

Iulia Bastys  
Chalmers University of Technology  
Gothenburg, Sweden  
bastys@chalmers.se

Frank Piessens  
Katholieke Universiteit Leuven  
Heverlee, Belgium  
Frank.Piessens@cs.kuleuven.be

Andrei Sabelfeld  
Chalmers University of Technology  
Gothenburg, Sweden  
andrei@chalmers.se

ABSTRACT
Recent years have seen a proliferation of research on information flow control. While the progress has been tremendous, it has also given birth to a bewildering breed of concepts, policies, conditions, and enforcement mechanisms. Thus, when designing information flow controls for a new application domain, the designer is confronted with two basic questions: (i) What is the right security characterization for a new application domain? and (ii) What is the right enforcement mechanism for a new application domain?

This paper puts forward six informal principles for designing information flow security definitions and enforcement mechanisms: attacker-driven security, trust-aware enforcement, separation of policy annotations and code, language-independence, justified abstraction, and permissiveness. We particularly highlight the core principles of attacker-driven security and trust-aware enforcement, giving us a rationale for deliberating over soundness vs. soundness. The principles contribute to roadmapping the state of the art in information flow security, weeding out inconsistencies from the folklore, and providing a rationale for designing information flow characterizations and enforcement mechanisms for new application domains.

CCS CONCEPTS
• Security and privacy → Formal methods and theory of security;

KEYWORDS
information flow control; attacker models; principles

ACM Reference Format:

1 INTRODUCTION
Information flow control tracks the flow of information in systems. It accommodates both confidentiality, when tracking information from secret sources (inputs) to public sinks (outputs), and integrity, when tracking information from untrusted sources to trusted sinks. The designer might struggle to select from the variety of mechanisms available. Information flow enforcement mechanisms have also been proposed in various styles and flavors, including static [20, 23, 79], dynamic [25, 26, 33], hybrid [14, 58], flow-(in)sensitive [41, 65], and language-(in)dependent [11, 24]. Further, some track pure data flows [72] whereas others also track control flow dependencies [67], adding to the complexity of choosing the right characterization.

Question 1. What is the right security characterization for a new application domain?

A number of information flow conditions has been proposed in the literature. For confidentiality, noninterference [22, 28], is a commonly advocated baseline condition stipulating that secret inputs do not affect public outputs. Yet noninterference comes in different styles and flavors: termination-(in)sensitive [67, 79], progress-(in)sensitive [3], and timing-sensitive [2], just to name a few. Other characterizations include epistemic [4, 35], quantitative [73], and conditions of information release [70], as well as weak [78], explicit [71], and observable [9] secrecy. Further, compositional security conditions [53, 61, 69] are often advocated, adding to the complexity of choosing the right characterization.

Question 2. What is the right enforcement mechanism for a new application domain?

While the progress has been tremendous, it has also given birth to a bewildering breed of concepts, policies, conditions, and enforcement mechanisms. These are often unconnected and ad-hoc, making it difficult to build on when developing new approaches. Thus, when designing information flow controls for a new application domain, the designer is confronted with two basic questions, for which there is no standard recipe in the literature.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-5993-1/18/10. ... $15.00  
https://doi.org/10.1145/3264820.3264824
We begin by presenting two core principles: attacker-driven security and trust-aware enforcement, followed by four additional principles. The core principles can be viewed as instantiations of the two broader principles on “defining threat models” and “defining the trusted computing base” [48, 56]. The instantiation to information flow control is non-trivially different from instantiations in other security areas, in particular in the case where trusted annotations are required on untrusted code.

**Principle 1** (Attacker-driven security). Security characterizations benefit from directly connecting to a behavioral attacker model, expressing (un)desirable behaviors in terms of system events that attackers can observe and trigger.

Key to this principle is a faithful attacker model, representing what events the attacker can observe and trigger. Focusing on attacker-driven security enables a systematic way to view the rich area of information flow characterizations. Figure 1 depicts a bird’s-eye view. The common attacker-driven conditions, such as the above-mentioned noninterference [22, 28] and epistemic security [4, 35], appear on the upper right. For systems that interact with an outside environment, it is important to model input/output behavior and its security implications. In this space, attacker-driven security is captured by so-called progress-sensitive security [57, 63, 64], in contrast to progress-insensitive security [3] that ignores leaks due to computation progress.

Throughout the paper, we will leverage the JSFlow [38] tool to illustrate the principles on JavaScript code fragments. We use high and low labels for secret and public data, respectively. JSFlow is a JavaScript interpreter that tracks information flow labels. JSFlow constructor `lbl` is used for assigning a high label to a value. As is common, JSFlow accommodates information release via declassification [70]. Primitive `declassify` is used for declassifying a value from high to low. Primitive `print` is used for output. We consider `print` statements to be public.

**Example 1** (Based on Program 2 [3]).

```javascript
i = 0;
while (i < Number.MAX_VALUE) {
    print(i);
    if (i == secret) { while (true) {} }
    i = i + 1;
}
```

In the above example, if the attacker is assumed to be able to observe the intermediate outputs of the computation, then the program is progress-sensitive insecure, otherwise is progress-insensitive secure. As JSFlow enforces progress-insensitive noninterference, it will accept the program.

Attacker-driven security is also represented by relaxations of noninterference to quantitative information flow [73] and information release [70], capturing scenarios of intended information release.

**Example 2** (Simple password checking [70]).

```javascript
guess = lbl(getUserInput());
result = declassify(guess == pwd);
```

The above example checks whether the user input retrieved via function `getUserInput()` matches the stored password `pwd`. The user input and variable `pwd` are assumed to be high, and result to be low, as an attacker should be only allowed to learn whether the user’s guess matches the stored password, but not the actual guess, nor the actual password.

When the attacker model combines confidentiality and integrity, their interplay requires careful treatment. For example, the goal of robust declassification [83] is to prevent untrusted data from affecting declassification decisions.

Further relaxations of noninterference bring us to soundness, inspired by a recent movement in the program analysis community. In their manifesto, Livshits et al. advocate soundness [51] of program analysis, arguing that it is virtually impossible to establish soundness for practical whole program analysis. While soundness breaks soundness, its goal is to explain and limit the implications of unsoundness.

In this sense, popular relaxations of noninterference like termination-insensitive [67, 79] and progress-insensitive [3] noninterference are soundness. Termination- and progress-insensitive conditions are often used to justify permissive handling of loops that branch on secrets by enforcement. However, this justification alone would exclude these conditions from being attacker-driven, unless the impact of unsoundness with respect to a behavioral attacker is characterized. Indeed, limiting implications of unsoundness for these conditions have been studied, e.g., by giving quantitative bounds on how much is leaked via the termination and progress channels [3].

The conditions of observable [9], weak [78], and explicit [71] secrecy are depicted in the lower right of Figure 1. These conditions are fundamentally different from attacker-driven definitions, clearly falling into the category of soundness. Rather than characterizing an attacker, they are tailored to describe properties of enforcement, catering to mechanisms like taint tracking [72]; pure data dependency analysis that ignores leaks due to control flow, and its enhancements with so-called observable [9] implicit flow checks.

Finally, in contrast to attacker-driven definitions, we distinguish verification conditions, such as those provided by compositional security [53, 61, 69], invariants [62], and unwinding conditions [29]. We bring up verification conditions in order to point out that they are not suitable to be used as definitions of security. Indeed, while compositional security is essential for scaling the reasoning about security enforcement [48, 52], compositionality per se is inconsequential for characterizing security against a concrete attacker [39]. We thus argue that it is valuable to aim at compositional verification conditions, as long as they are sufficient for implying security against a clearly specified attacker-driven characterization. The verification
While user-provided inputs are not. In this scenario, taint tracking (justifying the soundness of the verification conditions).

The execution of the program, but accepts it otherwise.

By fully-fledged information flow control. If

\[ k \]

is accepted by observable tracking, as \( k \) is never output, but rejected by fully-fledged information flow control. If \( h \) is \( true \), JSFlow blocks the execution of the program, but accepts it otherwise.

**Example 3** (Based on Program 3 [9]).

```plaintext
l = true;
k = true;
if (h) {l = false;}
if (l) {k = false;}
print(42);
```

While the above example encodes the value of high variable \( h \) into variable \( k \) through observable implicit flows, the program is accepted by observable tracking, as \( k \) is never output, but rejected by fully-fledged information flow control. If \( h \) is \( true \), JSFlow blocks the execution of the program, but accepts it otherwise.

**Principle 2** (Trust-aware security enforcement). Security enforcement benefits from explicit trust assumptions, making clear the boundary between trusted and untrusted computing base and guiding the enforcement design in accord.

Figure 1 illustrates this principle by listing the different enforcement mechanisms in the order of what code it is suitable for: from untrusted to trusted. This order loosely aligns untrusted code with attacker-driven security and trusted code with soundness. The rationale is that security enforcement for untrusted code needs to cover flows with respect to a given attacker-driven security, as the attacker has control over which flows to try to exploit. In contrast, trusted code can be harder to exploit. For example, in a scenario of injection attacks on a web server, the code is trusted while user-provided inputs are not. In this scenario, taint tracking is often sufficient, because the code does not contain malicious patterns that exploit control flows to mount attacks [72]. In other scenarios with trusted code, it is possible to establish security by a lightweight combination of an explicit-flow and graph-pattern analyses [66]. Overall, the permissiveness of mechanisms increases with the degree of trust to the code.

Trade-offs between taint tracking and fully-fledged information flow control have been subject to empirical studies [46]. The middle ground between tracking explicit and some implicit flows has been explored in implementations [9, 77] and formalizations [9] via observable tracking [9] that disregards control flows in the branches that are not taken by a monitoring mechanism.

**Example 3** (Based on Program 3 [9]).

```plaintext
l = true;
k = true;
if (h) {l = false;}
if (l) {k = false;}
print(42);
```

While the above example encodes the value of high variable \( h \) into variable \( k \) through observable implicit flows, the program is accepted by observable tracking, as \( k \) is never output, but rejected by fully-fledged information flow control. If \( h \) is \( true \), JSFlow blocks the execution of the program, but accepts it otherwise.

While the permissiveness of mechanisms generally increases with the degree of trust to the code, there is need for a systematic approach on choosing the right enforcement. We bring up two important aspects: (i) considerations of integrity and (ii) terminology inconsistencies.

For the integrity aspect, some literature doubts the importance of implicit flows for integrity. For example, Haack et al. suggest that "somehow implicit flows seem to be less of an issue for integrity requirements" [34]. To understand the root of the problem, it is fruitful to consider that integrity has different facets: integrity via invariance and via information flow [18]. The former is generally about safety properties, from data and predicate invariance to program correctness. It is often sufficient to enforce this facet of integrity with invariant checks and/or taint tracking (e.g., ensuring that tainted data has been sanitized before output). On the other hand, the latter is dual to confidentiality. Thus, implicit flows cannot be ignored for the information flow facet of integrity. Examples of implicit flows that matter for integrity (and forms of availability) are the inputs of coma [21] and crashed regular expression matching [80], where trusted code is fed untrusted inputs with the goal of corrupting the execution.

Interestingly, tainting and information flow tracking are sometimes used interchangeably in the literature, making it unclear what type of dependencies is actually tracked. For example, "information flow" approaches to Android app security are often taint trackers that do not track implicit flows [20, 25, 30]. Conversely a "taint tracker" for JavaScript is actually a mechanism that also tracks observable implicit flows [77]. In this paper, we distinguish between fully-fledged information flow tracking of both explicit and implicit flows versus taint tracking that only tracks explicit flows.

Trust-aware enforcement accommodates systematic selection of enforcement. Trusted, non-malicious, code with potentially untrusted inputs can be subject to vulnerability detection techniques like taint tracking. Untrusted, potentially malicious code, is subject to a more powerful analysis that takes into account attacker capabilities in a given runtime environment. Other considerations, like particular trust assumptions of a target domain and whether enforcement is decentralized, further affect the choice of trust-aware enforcement.

We discuss further prudent principles of general flavor, from the perspective of applying them to information flow control.
Principle 3 (Separation of policy annotations and code). Security policy annotations and code benefit from clear separation, especially when the policy is trusted and code is untrusted.

This principle governs syntactic policies as expressed by developers for a given program in terms of security labels, declassification annotations, and similar. We illustrate this principle on policies for information release, or declassification, using dimensions of declassification, with respect to what information is declassified, where (in the code), when (at what point of execution) and by whom (by what principal) [70].

The where dimension of declassification is concerned with policies that limit information release to specially marked locations in code (with declassification annotations). The principle implies that code annotated with declassification policies (e.g., [4, 10]) cannot be part of purely untrusted code, where the attacker can abuse annotations to release more information than intended.

If code of Example 2 were untrusted, an attacker could place the declassification annotation on the password pvd, and not on the result of equating pwd with the user input:

Example 4.

```
result = declassify(pwd);
```

In a case like this, there is need to strengthen declassification policies with other dimensions, such as what, when, and by whom, all specified separately from untrusted code.

Other cases such as delimited release [68] specify an external security policy via “escape hatches”, separating policy from code. At the same time, type systems for delimited release [68] can still allow declassify statements inside the syntax to help the program analysis accept the code. Programs with overly liberal declassification statements will be then rejected, as they are unsound with respect to external escape hatches. Since release of information is allowed only through the escape hatch expressions mentioned in the policy, declassifications as in Example 5 are accepted, while declassifications as in Examples 4 and 6 are not. JSFlow will accept all three snippets, as the monitor enforces only the where dimension of declassification.

Example 5 (Based on Example 1 (Avg) [68]).

```
avg = declassify((h_1 + \ldots + h_n)/n);
```

Example 6 (Based on Example 1 (Avg-Attack) [68]).

```
h_1 \neq h_i; \ldots h_n = h_i;
avg = declassify((h_1 + \ldots + h_n)/n);
```

Principle 3 is related to the previous principle of trust-aware enforcement, in the sense that an enforcement mechanism that relies on annotations needs to have strong assurance that the integrity of these annotations can be trusted, i.e. that they cannot be provided by the attacker in the form of annotated untrusted code, and that the execution engine can be trusted to preserve the integrity of the annotations.

Principle 4 (Language-independence). Language-independent security conditions benefit from abstracting away from the constructs of the underlying language. Language-independent enforcement benefits from simplicity and reuse.

While the challenges in information flow enforcement are often in the details of handling rich language constructs, these constructs are often inconsequential to the actual security. It is thus prudent to formulate security in an end-to-end fashion, on “macroflows” between sources and sinks, thus focusing on the interaction of the system with the environment, rather than on “microflows” between language constructs.

This principle tightly connects to Principle 1 on attacker-driven security. It also has beneficial implications for enforcement. For example, secure multi-execution [24] ensures security by executing a program multiple times, one run per security level, while carefully dispatching inputs and outputs to the runs with sufficient access rights. The elegance of secure multi-execution is its blackbox, language-independent, view of a system. This enables information flow control mechanisms like FlowFox [31] for the complex language of JavaScript, sidestepping a myriad of problems such as dynamic code evaluation, type coercion, scope, and sensitive upgrade [6, 82], which challenge JavaScript-specific information flow trackers [15, 37]. Language-independence makes FlowFox more robust to changes in the JavaScript standards.

Recall Example 3. Its execution is blocked by JSFlow when h is true, but accepted otherwise. In contrast, FlowFox produces the low output irrespective of the value of h.

Faceted values [7] show that ideas from information flow control and secure multi-execution can be combined in a single mechanism.

Principle 5 (Justified abstraction). The level of abstraction in the security model benefits from reflecting attacker capabilities.

Also connecting to Principle 1, this principle focuses on the level of abstraction that is adequate to model a desired attack surface. It relates to “integrative pluralism” [74] and not relying on a single ontology in the quest for the Science of Security. It also relates to the problems with “provable security” [40], when security is proved with respect to an abstraction that ignores important classes of attacks. Thus, it is important to reflect attacker capabilities in the attacker model and provide a strong connection between concrete and abstract attacks.

A popular line of work is on information flow control for timing attacks [47]. Timing is often modeled by timing cost labels [2] in the semantics. However, modeling time in a high-level language places demands on carrying the assumptions over to low-languages and hardware, as to take into account low-level attacks, for example, via data and instruction cache [75]. Thus, this principle emphasizes low-level security models that reflect attackers’ observations of time. Mantel and Starostin study the effects of non-justified timing abstractions on multiple security-establishing program transformations [54].

Example 7.

```
if (h == 1) { h’ = h_1; }
else { h’ = h_2; }
```

An attacker capable of analyzing the time it takes to execute the snippet above can infer information about the secret h. The execution time will be shorter if h = 1, as the value of h_1 will already be present in the cache by the time the last assignment is performed.
The program is accepted by JSFlow, as it does not assume such attackers.

Principle 5 is particularly important for security-critical systems, where even a low bandwidth of leaks can be devastating. For example, information flow analysis for VHDL by Tolstrup et al. [76] is in line with this principle by faithfully modeling time at circuit level. Zhang et al. [84] propose a hardware design language SecVerilog and prove that it enforces timing-sensitive noninterference. Work on blackbox timing mitigation for web application by Askarov et al. [5] is also interesting in this space. Their blackbox mechanism relies on no high-level abstractions of time because mitigation is performed on the endpoints of the system. The timing leak bandwidth is controlled by appropriately delaying attacker-observable events.

**Principle 6 (Permissiveness).** Enforcement for untrusted code particularly benefits from reducing false negatives (soundness), while enforcement for trusted code particularly benefits from reducing false positives (high permissiveness).

This principle further elaborates consequences of treating untrusted and trusted code. While it is crucial to provide coverage against attacks by untrusted code (soundness), for trusted code the focus is on reducing false alarms (high permissiveness). Indeed, it makes sense to prioritize security for potentially malicious code and to prioritize reducing false alarms for trusted code. The latter is a key consideration for adopting vulnerability detection tools by developers.

Consider again the program in Example 3. While a false positive for a fully-fledged information flow tracker such as JSFlow, the snippet is accepted by both observable flow and taint trackers.

It is interesting to apply Principle 6 to the setting of Android apps, a typical setting of potentially malicious code. Currently, the state of the art is largely taint tracking mechanisms like TaintDroid [25], DroidSafe [30], and HornDroid [20], failing to detect implicit flows [27]. Interestingly, there is evidence of implicit flows in malicious code on the web [42]. We anticipate implicit flows to be exercised by malicious Android apps whenever there arises a need to bypass explicit flow checks. Thus, we project a trend for taint trackers in this domain to be extended into fully-fledged information flow trackers, with first steps in this direction already being made [81].

### 3 RELATED WORK

Our principles draw inspiration from Abadi and Needham’s inform principles for designing cryptographic protocols [1].


### 4 CONCLUSION

We have presented prudent principles for designing information flow control for emerging domains. The core principles of attacker-driven security and trust-aware enforcement provide a rationale for deliberating over soundness vs. soundness, while the additional principles of separation of security policies from code, language-independent security conditions, justified abstraction, and permissiveness help design information flow control characterizations and enforcement mechanisms.

**Acknowledgments.** This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. This work was also partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

**REFERENCES**


