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. soundiness. 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:
Iulia Bastys, Frank Piessens, and Andrei Sabelfeld. 2018. Prudent Design Principles for Information Flow Control. In *The 13th Workshop on Programming Languages and Analysis for Security (PLAS'18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3264820.3264824

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.

Motivation. Recent years have seen a proliferation of research on information flow control [16, 17, 19, 39, 49, 55, 67, 70, 72, 73], leading to applications in a wide range of areas including hardware [8], operating system microkernels [59] and virtualization platforms [32], programming languages [36, 37], mobile operating systems [44], web browsers [12, 43], web applications [13, 45], and distributed systems [50]. A recent special issue of Journal of Computer Security on verified information flow [60] reflects an active state of the art.

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.

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?

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 enforcement mechanism.

Contributions. This paper puts forward principles for designing information flow security definitions and enforcement mechanisms. The goal of the principles is to help 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 mechanisms for new application domains.

The rationale rests on the following principles: *attacker-driven security*, *trust-aware enforcement*, *separation of policy annotations*

PLAS'18, October 19, 2018, Toronto, ON, Canada
© 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

and code, *language-independence*, *justified abstraction*, and *permissiveness*.

Scope. Given the area's maturity, this work is deliberately not a literature survey. There are several excellent surveys overviewing different aspects of information flow security [16, 17, 19, 39, 49, 55, 67, 70, 72, 73], further discussed in Section 3. Rather, we seek to empower information flow control mechanism designers by illuminating key principles we believe are important when designing new mechanisms.

2 DESIGN PRINCIPLES

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])**.**

```
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])**.**

```
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 soundiness, inspired by a recent movement in the program analysis community. In their manifesto, Livshits et al. advocate *soundiness* [51] of program analysis, arguing that it is virtually impossible to establish soundness for practical whole program analysis. While soundiness 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 soundiness. 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 soundiness. 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 compositionality 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

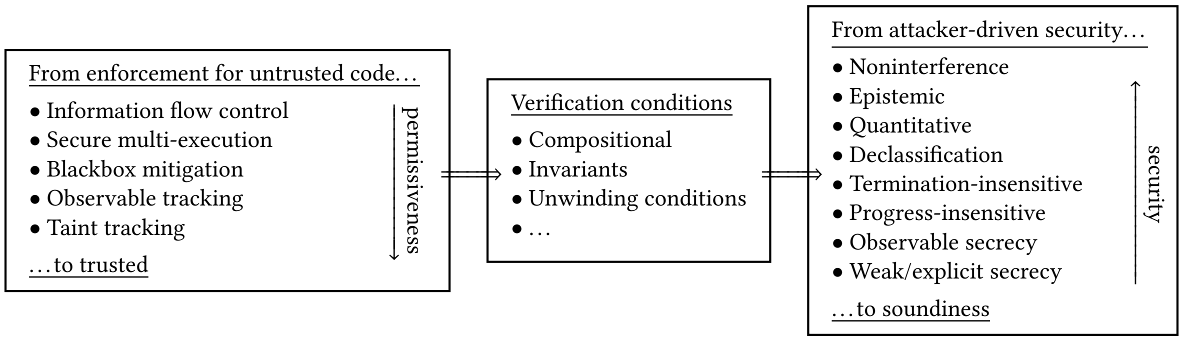

Figure 1: Bird's-eye view: enforcement, verification conditions, and security characterizations

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 conditions are depicted in the middle of the figure. The arrows between the boxes illustrate logical implication, from enforcement to verification conditions (justifying the usefulness of verification conditions) and from verification conditions to security conditions (justifying the soundness of the verification conditions).

> **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]).

```
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 pwd, 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((h₁ + ... + hₙ)/n);
```

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

```
h₁ = hᵢ; ... hₙ = hᵢ;
avg = declassify((h₁ + ... + hₙ)/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] enforces 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′ = h₁; }
else { h′ = h₂; }
h′ = h₁;
```

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 informal principles for designing cryptographic protocols [1].

Prior work has focused on different aspects of information flow security. Sabelfeld and Myers [67] roadmap language-based information security definitions and static enforcement mechanisms. Le Guernic [49] overviews dynamic techniques. Sabelfeld and Sands [70] outline principles and dimensions of declassification, roadmapping the area of intended information release. Smith [73] gives an account of foundations for quantitative information flow. Schwartz et al. [72] survey dynamic taint analysis and symbolic execution for security. Hedin and Sabelfeld [39] give a uniform presentation of dominant security conditions by gradually refining the indistinguishability relation that models the attacker. Bielova [16] roadmaps JavaScript security policies and their enforcement in a web browser. Mastroeni [55] gives an overview of information flow techniques based on abstract interpretation. Broberg et al. [19] give a systematic view of dynamic information flow. Bielova and Rezk [17] provide a rigorous taxonomy of information flow monitors. A recent special issue of Journal of Computer Security [60] showcases a current snapshot of work on verified information flow.

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. soundiness, 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

[1] Martín Abadi and Roger M. Needham. 1996. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Trans. Software Eng.* 22, 1 (1996), 6–15.

[2] Johan Agat. 2000. Transforming Out Timing Leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2000, Boston, MA, USA, January 19-21, 2000.* ACM, 40–53.

[3] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Computer Security - ESORICS 2008 - 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings (Lecture Notes in Computer Science)*, Vol. 5283. Springer, 333–348.

[4] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *28th IEEE Symposium on Security and Privacy, S&P 2007, Oakland, CA, USA, May 20-23, 2007.* IEEE Computer Society, 207–221.

[5] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive blackbox mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, IL, USA, October 4-8, 2010.* ACM, 297–307.

[6] Thomas H. Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009.* ACM, 113–124.

[7] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, PA, USA, January 22-28, 2012.* ACM, 165–178.

[8] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew

Tolmach. 2016. A verified information-flow architecture. *Journal of Computer Security* 24, 6 (2016), 667–688.

[9] Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. 2017. We Are Family: Relating Information-Flow Trackers. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017. Proceedings (Lecture Notes in Computer Science)*, Vol. 10492. Springer, 124–145.

[10] Gilles Barthe, Salvador Cavadini, and Tamara Rezk. 2008. Tractable Enforcement of Declassification Policies. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, PA, USA, 23-25 June, 2008*. IEEE Computer Society, 83–97.

[11] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. 2012. Secure Multi-Execution through Static Program Transformation. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings (Lecture Notes in Computer Science)*, Vol. 7273. Springer, 186–202.

[12] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, CA, USA, February 8-11, 2015*. The Internet Society.

[13] Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. 2016. CoSMed: A Confidentiality-Verified Social Media Platform. In *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings (Lecture Notes in Computer Science)*, Vol. 9807. Springer, 87–106.

[14] Frédéric Besson, Nataliia Bielova, and Thomas P. Jensen. 2013. Hybrid Information Flow Monitoring against Web Tracking. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium, CSF New Orleans, LA, USA, 26-28 June, 2013*. IEEE Computer Society, 240–254.

[15] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Information Flow Control in WebKit's JavaScript Bytecode. In *Principles of Security and Trust - 3rd International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Vol. 8414. Springer, 159–178.

[16] Nataliia Bielova. 2013. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *J. Log. Algebr. Program.* 82, 8 (2013), 243–262.

[17] Nataliia Bielova and Tamara Rezk. 2016. A Taxonomy of Information Flow Monitors. In *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Vol. 9635. Springer, 46–67.

[18] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. 2010. Unifying Facets of Information Integrity. In *Information Systems Security - 6th International Conference, ICISS 2010, Gandhinagar, India, December 17-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Vol. 6503. Springer, 48–65.

[19] Niklas Broberg, Bart van Delft, and David Sands. 2015. The Anatomy and Facets of Dynamic Policies. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. IEEE Computer Society, 122–136.

[20] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 2016. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 47–62.

[21] Richard M. Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. 2009. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, NY, USA, 8-10 July, 2009*. IEEE Computer Society, 186–199.

[22] Ellis S. Cohen. 1977. Information Transmission in Computational Systems. In *Proceedings of the Sixth Symposium on Operating System Principles, SOSP 1977, Purdue University, West Lafayette, Indiana, USA, November 16-18, 1977*. ACM, 133–139.

[23] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (1977), 504–513.

[24] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, Oakland, CA, USA, May 16-19, 2010*. IEEE Computer Society, 109–124.

[25] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. USENIX Association, 393–407.

[26] Jeffrey S. Fenton. 1974. Memoryless Subsystems. *Comput. J.* 17, 2 (1974), 143–147.

[27] Christian Fritz, Steven Arzt, and Siegfried Rasthofer. 2018. DroidBench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android. https://github.com/secure-software-engineering/DroidBench.

[28] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, S&P 1982, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 11–20.

[29] Joseph A. Goguen and José Meseguer. 1984. Unwinding and Inference Control. In *1984 IEEE Symposium on Security and Privacy, S&P 1984, Oakland, CA, USA, April 29 - May 2, 1984*. IEEE Computer Society, 75–87.

[30] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, CA, USA, February 8-11, 2015*. The Internet Society.

[31] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, October 16-18, 2012*. ACM, 748–759.

[32] Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. 2016. Provably secure memory isolation for Linux on ARM. *Journal of Computer Security* 24, 6 (2016), 793–837.

[33] Gurvan Le Guernic. 2007. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF 2007, Venice, Italy, 6-8 July, 2007*. IEEE Computer Society, 218–232.

[34] Christian Haack, Erik Poll, and Aleksy Schubert. 2009. Explicit information flow properties in JML. *Proc. WISSEC* (2009).

[35] Joseph Y. Halpern and Kevin R. O'Neill. 2008. Secrecy in Multiagent Systems. *ACM Trans. Inf. Syst. Secur.* 12, 1 (2008), 5:1–5:47.

[36] Christian Hammer and Gregor Snelting. 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.* 8, 6 (2009), 399–422.

[37] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. 2016. Information-flow security for JavaScript and its APIs. *Journal of Computer Security* 24, 2 (2016), 181–234.

[38] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*.

[39] Daniel Hedin and Andrei Sabelfeld. 2012. A Perspective on Information-Flow Control. In *Software Safety and Security*. NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 33. IOS Press, 319–347.

[40] Cormac Herley and Paul C. van Oorschot. 2017. SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit. In *38th IEEE Symposium on Security and Privacy, S&P 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 99–120.

[41] Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, SC, USA, January 11-13, 2006*. ACM, 79–90.

[42] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, IL, USA, October 4-8, 2010*. ACM, 270–283.

[43] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2012. Establishing Browser Security Guarantees through Formal Shim Verification. In *Proceedings of the 21th USENIX Security Symposium, USENIX Security 12, Bellevue, WA, USA, 8-10 August, 2012*. USENIX Association, 113–128.

[44] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-Time Enforcement of Information-Flow Properties on Android - (Extended Abstract). In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings (Lecture Notes in Computer Science)*, Vol. 8134. Springer, 775–792.

[45] Sudeep Kanav, Peter Lammich, and Andrei Popescu. 2014. A Conference Management System with Verified Document Confidentiality. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science)*, Vol. 8559. Springer, 167–183.

[46] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. 2008. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings (Lecture Notes in Computer Science)*, Vol. 5352. Springer, 56–70.

[47] Boris Köpf and David A. Basin. 2006. Timing-Sensitive Information Flow Analysis for Synchronous Systems. In *Computer Security - ESORICS 2006 - 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006. Proceedings (Lecture Notes in Computer Science)*, Vol. 4189. Springer, 243–262.

[48] Carl E. Landwehr, Dan Boneh, John C. Mitchell, Steven M. Bellovin, Susan Landau, and Michael E. Lesk. 2012. Privacy and Cybersecurity: The Next 100 Years. *Proc. IEEE* 100, Centennial-Issue (2012), 1659–1673.

[49] Gurvan Le Guernic. 2007. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. Ph.D. Dissertation. Kansas State University. http:

//tel.archives-ouvertes.fr/tel-00198621/fr/

[50] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. 2017. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security* 25, 4-5 (2017), 367–426.

[51] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[52] Heiko Mantel. 2002. On the Composition of Secure Systems. In *23rd IEEE Symposium on Security and Privacy, S&P 2002, Oakland, CA, USA, May 12-15, 2002*. IEEE Computer Society, 88–101.

[53] Heiko Mantel, David Sands, and Henning Sudbrock. 2011. Assumptions and Guarantees for Compositional Noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society, 218–232.

[54] Heiko Mantel and Artem Starostin. 2015. Transforming Out Timing Leaks, More or Less. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015. Proceedings (Lecture Notes in Computer Science)*, Vol. 9326. Springer, 447–467.

[55] Isabella Mastroeni. 2013. Abstract interpretation-based approaches to Security - A Survey on Abstract Non-Interference and its Challenging Applications. *arXiv preprint arXiv:1309.5131* 129 (2013), 41–65.

[56] Gary McGraw and J. Gregory Morrisett. 2000. Attacking Malicious Code: A Report to the Infosec Research Council. *IEEE Software* 17, 5 (2000), 33–41.

[57] Scott Moore, Aslan Askarov, and Stephen Chong. 2012. Precise enforcement of progress-sensitive security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, October 16-18, 2012*. ACM, 881–893.

[58] Scott Moore and Stephen Chong. 2011. Static Analysis for Efficient Hybrid Information-Flow Control. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society, 146–160.

[59] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *34th IEEE Symposium on Security and Privacy, S&P 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, San Francisco, CA, 415–429. https://doi.org/10.1109/SP.2013.35

[60] Toby C. Murray, Andrei Sabelfeld, and Lujo Bauer. 2017. Special issue on verified information flow security. *Journal of Computer Security* 25, 4-5 (2017), 319–321.

[61] Toby C. Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. 2016. Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 417–431.

[62] David A. Naumann. 2006. From Coupling Relations to Mated Invariants for Checking Information Flow. In *Computer Security - ESORICS 2006 - 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006. Proceedings (Lecture Notes in Computer Science)*, Vol. 4189. Springer, 279–296.

[63] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. 2006. Information-Flow Security for Interactive Programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop, CSFW 2006, Venice, Italy, 5-7 July, 2006*. IEEE Computer Society, 190–201.

[64] Willard Rafnsson, Deepak Garg, and Andrei Sabelfeld. 2016. Progress-Sensitive Security for SPARK. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings (Lecture Notes in Computer Science)*, Vol. 9639. Springer, 20–37.

[65] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, 17-19 July, 2010*. IEEE Computer Society, 186–199.

[66] Alejandro Russo, Andrei Sabelfeld, and Keqin Li. 2010. Implicit flows in malicious and nonmalicious code. In *Logics and Languages for Reliability and Security*. NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 25. IOS Press, 301–322.

[67] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[68] Andrei Sabelfeld and Andrew C. Myers. 2003. A Model for Delimited Information Release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*. 174–191.

[69] Andrei Sabelfeld and David Sands. 2000. Probabilistic Noninterference for Multi-Threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, 3-5 July, 2000*. IEEE Computer Society, 200–214.

[70] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *Journal of Computer Security* 17, 5 (2009), 517–548.

[71] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. 2016. Explicit Secrecy: A Policy for Taint Tracking. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 15–30.

[72] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, Oakland, CA, USA, May 16-19, 2010*. IEEE Computer Society, 317–331.

[73] Geoffrey Smith. 2009. On the Foundations of Quantitative Information Flow. In *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Vol. 5504. Springer, 288–302.

[74] Jonathan M. Spring, Tyler Moore, and David J. Pym. 2017. Practicing a Science of Security: A Philosophy of Science Perspective. In *NSPW*. ACM, 1–18.

[75] Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. 2013. Eliminating Cache-Based Timing Attacks with Instruction-Based Scheduling. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings (Lecture Notes in Computer Science)*, Vol. 8134. Springer, 718–735.

[76] Terkel K. Tolstrup, Flemming Nielson, and Hanne Riis Nielson. 2005. Information Flow Analysis for VHDL. In *Parallel Computing Technologies, 8th International Conference, PaCT 2005, Krasnoyarsk, Russia, September 5-9, 2005, Proceedings (Lecture Notes in Computer Science)*, Vol. 3606. Springer, 79–98.

[77] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *14th Annual Network and Distributed System Security Symposium, NDSS 2007, San Diego, CA, USA, February 28 - March 2, 2007*. The Internet Society.

[78] Dennis M. Volpano. 1999. Safety versus Secrecy. In *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings (Lecture Notes in Computer Science)*, Vol. 1694. Springer, 303–311.

[79] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2/3 (1996), 167–188.

[80] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. 2017. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 10206. Springer, 3–20.

[81] Wei You, Bin Liang, Jingzhe Li, Wenchang Shi, and Xiangyu Zhang. 2015. Android Implicit Information Flow Demystified. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*. ACM, 585–590.

[82] Stephan Zdancewic. 2002. *Programming Languages for Information Security*. Ph.D. Dissertation. Cornell University.

[83] S. Zdancewic and A. C. Myers. 2001. Robust Declassification. In *Computer Security Foundations Workshop*.

[84] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. ACM, 503–516.