CHALMERS
UNIVERSITY OF TECHNOLOGY

LazyTAP: On-Demand Data Minimization for Trigger-Action Applications

Citation for the original published paper (version of record):
Ahmadpanah, S., Hedin, D., Sabelfeld, A. (2023). LazyTAP: On-Demand Data Minimization for Trigger-Action Applications. Proceedings - IEEE Symposium on Security and Privacy, Volume 2023-May: 3079-3097. http://dx.doi.org/10.1109/SP46215.2023.00147

LazyTAP: On-Demand Data Minimization for Trigger-Action Applications

Mohammad M. Ahmadpanah*, Daniel Hedin*,†, and Andrei Sabelfeld*

*Chalmers University of Technology
†Mälardalen University

***Abstract*—Trigger-Action Platforms (TAPs) empower applications (apps) for connecting otherwise unconnected devices and services. The current TAPs like IFTTT require trigger services to push excessive amounts of sensitive data to the TAP regardless of whether the data will be used in the app, at odds with the principle of *data minimization*. Furthermore, the rich features of modern TAPs, including IFTTT *queries* to support multiple trigger services and *nondeterminism* of apps, have been out of the reach of previous data minimization approaches like minTAP. This paper proposes LazyTAP, a new paradigm for fine-grained *on-demand data minimization*. LazyTAP breaks away from the traditional *push-all* approach of coarse-grained data over-approximation. Instead, LazyTAP *pulls* input data on-demand, once it is accessed by the app execution. Thanks to the fine granularity, LazyTAP enables tight minimization that naturally generalizes to support multiple trigger services via queries and is robust with respect to nondeterministic behavior of the apps. We achieve seamlessness for third-party app developers by leveraging laziness to defer computation and proxy objects to load necessary remote data behind the scenes as it becomes needed. We formally establish the correctness of LazyTAP and its minimization properties with respect to both IFTTT and minTAP. We implement and evaluate LazyTAP on app benchmarks showing that on average LazyTAP improves minimization by 95% over IFTTT and by 38% over minTAP, while incurring a tolerable performance overhead.**

I. Introduction

Trigger-Action Platforms (TAPs) like IFTTT ("If This Then That") [33], Zapier [54], and Microsoft Power Automate [43] excel at connecting otherwise unconnected devices and services. Consider services that manage users' data like Google Calendar for calendar appointments and Trakt for keeping track of TV shows and movies watched. TAPs enable popular automation applications (or *apps*) like "Every morning at 7am, send a Slack message with the first meeting of the day from Google Calendar" [29] (app *B* among our running examples) or "When you turn your Samsung TV on after 5pm on Saturdays, pick one of the personalized movie recommendations from Trakt" [37] (app *J* among our running examples). In these examples, the TAP gets the initial app inputs from *trigger* services (Time and Samsung TV), requests further inputs from *query* services (from Google Calendar and Trakt), and sends the outputs to *action* services (Slack and Notification).

Privacy concerns. With the convenience and interoperability of TAPs comes the concern that the TAP is effectively a "person-in-the-middle", acting on behalf of the user with respect to trigger and action services. This poses a privacy challenge since in the event of a compromised TAP, the users' sensitive input data is also compromised [10], [9], [15], [1], [14].

The current practices of TAPs like IFTTT inherently rely on the *push-all* approach for input data. When a new event is emitted by the trigger service, all input data attributes are indiscriminately pushed to the TAP, regardless of whether the data will be used in the app execution. This coarse-grained over-approximation is at odds with *data minimization*, a principle stipulating to limit the data to "what is necessary in relation to the purposes for which they are processed" [21]. This important principle is adopted by legal frameworks like the General Data Protection Regulation (GDPR) [21] and the California Privacy Rights Act (CPRA) [18].

Data minimization first of all implies minimizing the possibility of *accessing* personal data [45]. Next within the remaining possibilities, the amount of personal data that is *stored* should be minimized. Finally, the *time* of storing sensitive data should also be minimized. Our work focuses on the first, most desired type of minimization: *data-access minimization*. This privacy goal, in line with previous work on data minimization on TAPs [14], is appealing because it is robust with respect to potential data breaches on TAPs. Indeed, TAPs not always succeed to safeguard user data received from trigger services [1].

From triggers to queries. The push-all approach exacerbates the privacy problem in the presence of multiple sources of input data. IFTTT allows multiple inputs via the mechanism of *queries* [35], [38], a recently introduced feature for paid users to create, publish, and run apps with additional data sources. Kalantari et al. [41] identify 90 sensitive queries for access to private data in categories that include health & fitness, communication, finance & payments, voice assistants, security & monitoring, cloud storage, photo & video, connected car, and contacts.

In app *B*, the trigger service is Time, triggering the app at 7am every day. However, the sensitive input of the app is loaded by a query to Google Calendar. Even though the app needs information about only one meeting, the TAP excessively loads all attributes of all recent meetings. Moreover, even if the app only asks for a query conditionally on some input, a TAP like IFTTT will always load the most recent 50 query events [34] regardless of the input and whether the data

TAP	Minimization wrt	Minimization guarantees
IFTTT	None	Push all, no minimization guarantees
Static minTAP	Ill-intended TAP	Input-unaware minimization
Dynamic minTAP		Input-sensitive minimization wrt trigger input, No attributes when skip/timeout, No support for queries or nondeterminism
LazyTAP	TAP willing to minimize	Input-sensitive minimization wrt trigger and query inputs

Table I: Comparison of TAPs.

is necessary for the execution.

Similarly, in app *J*, all recommended movies will be excessively sent from Trakt to the TAP, letting the TAP make a randomized pick, even though only one movie is recommended to the user. This example also illustrates the use of *nondeterminism* in apps, which can stem from, for example, random number generation or reading wall time.

Data minimization in TAPs. As summarized in Table I, IFTTT follows a traditional push-all approach with no data minimization support with respect to trigger and query data.

Recent work introduces minTAP [14], also reflected in Table I. minTAP assumes an ill-intended TAP trying to break data minimization by maliciously manipulating the apps to extend access to sensitive data and by gaining access attributes beyond those that are necessary for the apps. The solution taken by minTAP is preprocessing [8] data on the trigger service, with the goal of not sending any redundant data to the TAP. This solution requires a trusted client outside of the TAP that performs attribute dependency analysis of the apps and ensures the integrity of installed apps.

Static minTAP performs static analysis of the app at the time of installation, identifies the necessary attributes and passes the information to trigger services per app, so that the redundant attributes are dropped upon trigger events. Static minTAP performs input-unaware minimization, unable to minimize attributes in cases when their use is dependent on runtime values. For app *B*, this implies always revealing meeting details even if the meeting does not meet an input-specific condition (e.g., the meeting location is the office), and for app *J*, this implies always revealing the full list of recommended movies even if only one movie needs to be recommended.

Dynamic minTAP pre-runs the app code in a sandbox on the trigger service to determine which trigger attributes are needed and drops the rest of the attributes. Dynamic minTAP thus performs input-sensitive minimization, gaining precision compared to static minTAP. Furthermore, the trigger service has the possibility of skipping events altogether.

Generally, minTAP does not support queries. Unfortunately, the dynamic solution is fundamentally limited with respect to queries. Indeed, the trigger service has no access to the data from the query services, which makes it impossible to be meaningfully input-sensitive when pre-running the app code. In app *B*, the Time trigger does not (and should not) have access to Google Calendar data. Similarly, the dynamic solution is fundamentally limited with respect to nondeterminism. Indeed, running the app code twice fails to reproduce the exact behavior of the app in the presence of nondeterminism. In app *J*, it is hard to guarantee that the outcome of random number generation will be the same in the pre-run on trigger service and in the actual run on the TAP. Nondeterminsm like wall-time reads and scheduling-related nondeterminism is a challenge to reproduce, even with access to seedable pseudo-random number generators.

LazyTAP to the rescue. We propose LazyTAP, a new paradigm for fine-grained *on-demand data minimization*. LazyTAP breaks away from the push-all approach of coarse-grained data over-approximation. Instead, LazyTAP *pulls* input data on-demand, whenever it is needed by the app execution. Thanks to the fine granularity, LazyTAP prevents sending the full calendar event stream and only sends the required attributes of at most one calendar event in the Slack message in app *B*, and only sends at most one movie recommendation in lieu of sending the whole movie recommendation array in app *J*.

LazyTAP assumes the TAP is incentivized to support the principle of data minimization for the sake of its users and in the light of the applicable legal frameworks. The possibility of driving minimization from the TAP itself enables us with a powerful possibility of pulling data on-demand. This paradigm is different from assuming the TAP intentionally tries to break data minimization [14], which necessitates preprocessing [8] the data before it is sent to the TAP, or encrypting data for distrusted TAPs [16], [15].

Compared to minTAP, LazyTAP does not require the trigger service to run the app, as there is only a single run of the app involved, and there is no need for a trusted client. As trigger data is pulled on the fly for a given run, LazyTAP guarantees input-sensitive minimization (see Table I). Further, LazyTAP supports queries and is robust with respect to nondeterminism while preserving the behavior of the underlying app, even in the presence of primitives like random number generation and wall-time reads. This implies sending precisely the meeting attributes used in the notification in app *B* and precisely the recommended movie in app *J*. The fact that LazyTAP seeks to help TAPs to provide data minimization guarantees makes it attractive for future adoption by TAPs, compared to solutions that focus on ill-intended TAPs.

The elegance of LazyTAP allows us to achieve seamlessness for app developers. Under the current TAPs, trigger and query data is received and stored in an object tree which is processed by app code. In contrast, LazyTAP creates so-called *remote objects* for trigger and query data that look to app code like local objects but are in fact populated by network requests to trigger and query services at runtime, and only

as much as needed by the given app execution. We develop a novel architecture for on-demand computation in third-party JavaScript apps that leverages laziness, in the form of deferred computation, and proxy objects to load necessary remote data behind the scenes as it becomes needed.

Moving from a push-all to a pull-on-demand paradigm on the TAP requires changes to the trigger/query services for compatibility reasons. These changes are straightforward and can be shimmed on the services themselves, or using third parties. Shimming on the services does not change the trust assumption on the services (which already hold user data) and is the natural choice (services must implement a compatibility layer with IFTTT already now [34]). Shimming by a third party is a last resort when modification of the service is not possible.

We formalize our approach, characterizing the correctness of LazyTAP and its minimization properties with respect to both IFTTT and minTAP. We prove that LazyTAP intrinsically improves minimization of input data in general with reference to any analysis in the style of minTAP.

The key idea of data-access minimization by on-demand computation is independent of the TAP and can be realized on various architectures. Yet for prototyping LazyTAP, we pick IFTTT as a starting point with the benefit of allowing a direct comparison with IFTTT and IFTTT-based implementation of minTAP.

Due to the novelty of queries and recently introduced paywalls for users to publish apps with queries, these apps are not frequently found among the public IFTTT apps. Yet our empirical analysis confirms that the published apps with queries do exhibit a broad range of data dependencies, beyond the reach of data minimization in minTAP.

We implement and evaluate LazyTAP on app benchmarks showing that on average LazyTAP improves minimization by 95% over IFTTT and by 38% over minTAP, while incurring a tolerable performance overhead.

The source code and benchmarks are available [2]. The implementation is readily deployable on AWS Lambda [4].

Contributions. The paper offers these contributions:

- We introduce LazyTAP, a new paradigm for on-demand computation on trigger-action platforms. Breaking away from the coarse-grained push-all approach, we enable pull-on-demand computation where data is pulled on demand at a fine level of granularity (Section III).
- We develop an architecture that leverages a novel combination of laziness in the form of deferred computations and proxy objects to pull remote data behind the scenes on a by-need basis, in a fashion seamless to app developers (Section III).
- We formally establish the correctness of LazyTAP and its minimization properties with respect to both IFTTT and minTAP (Section IV).
- We implement LazyTAP in a setting based on IFTTT and demonstrate by benchmarks that on average LazyTAP improves minimization by 95% over IFTTT and by 38% over minTAP (Section V).

II. Motivating examples

This section motivates the need for LazyTAP in comparison with IFTTT and minTAP by three examples of increasing complexity. As discussed in Section I, IFTTT has no minimization guarantees, always asking for all information from services, while minTAP precomputes the set of required attributes. LazyTAP goes further and leverages on-demand lazy computation to fetch only the attributes accessed during execution. The examples in this section are our running examples: the first two examples are the ones foreshadowed in Section I, and all three examples are a part of the benchmarks used in the detailed comparison in Section V, apps *B*, *J*, and *E*, respectively.

A. Threat model and assumptions

Prior to presenting the motivating examples, we recap the threat model and assumptions mentioned in Section I. The asset we protect in our setting is sensitive user data. Upon executing an installed app on a TAP, sensitive user data enters the TAP from trigger and query services. Our focus is on data-access minimization to limit the amount of data sent from trigger and query services to the TAP. Generally, data minimization seeks to mitigate the threat of data breaches resulting from attacks, human error, system failures, unauthorized access to personal information, data misuse or abuse, and non-compliance with privacy regulations [21], [45]. By limiting the amount of sensitive data accessed, TAPs thus reduce the potential impact of data breaches and limit the amount of sensitive information that can be compromised.

B. Calendar to Slack

The first example app (*B*) is an automation app for personal productivity. Every work day starts with a Slack notification reminding the first meeting from the user's calendar [29].

Figure 1: A meeting notification app.

Figure 1 illustrates the app structure. A certain time triggers the app that queries Google Calendar to extract today's important meetings. In case the first meeting occurs in the office, the Slack notification contains the title and the starting time of the meeting. Otherwise, the user will be notified that the first meeting's location is not the office.

In this app, the list of personal meetings is privacy-sensitive as the detail of the user's schedule is being shared with the

TAP. From a data minimization perspective, only the location of the first meeting along with the title and the starting time if it happens in office is necessary to operate the automation.

All the calendar events from the query are sent to IFTTT, which is excessive from the data minimization point of view. While dynamic minTAP fundamentally fails to support queries, static minTAP extended to support queries will over-approximate and report these attributes as necessary even if the meeting is not in the office. LazyTAP, on the other hand, requests the location of the first meeting and checks whether it is office. Then it asks for the title and time of the first event only if the else branch is taken.

C. Movie recommender

The second example is a simplification of an existing IFTTT app (*J*) that automates movie recommendation workflow according to user preferences [37]. Once the user turns on the Samsung Smart TV, a movie will be suggested to watch.

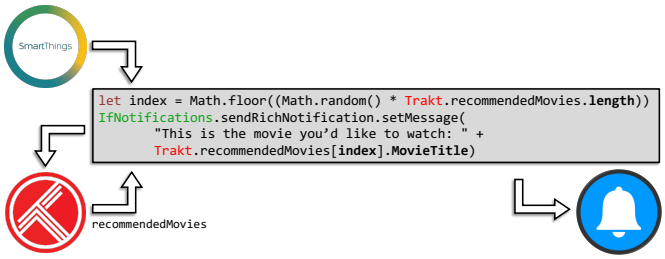

Figure 2: A movie recommender app.

As shown in Figure 2, the trigger is the TV being turned on. The app picks a random item from the personalized list of recommended movies based on the user's profile in Trakt, the platform that keeps track of user's watched movies and TV shows. Hence, the list of recommended movies is privacy-sensitive.

Data minimization demands sharing the minimum amount of necessary data with third-party entities like TAPs. This app only reads the number of movies and the title of the randomly selected movie from the list. IFTTT excessively sends the whole array of recommended movies.

Dynamic minTAP is inherently inapplicable for this example, not only due to the lack of support for queries, but also because the trigger service cannot predictably reproduce the outcome of random number generation of the app run on the TAP. Static minTAP cannot do better than IFTTT for this app either because no static approach is able to predict the random number by analyzing the app source code.

LazyTAP follows the data minimization principle by requesting the data attributes accessed by the execution on demand. Thus, only the number of recommended movies and only the movie title of the randomly picked element are sent to the TAP.

D. Parking space finder

The third example (*E*) is a parking space finder app facilitating tasks of a morning work routine. The user leaves home to commute to the meeting place by car. Once the door gets closed, the app looks for a parking area nearby the upcoming meeting's location and invokes the navigator on an Android phone with the parking location.

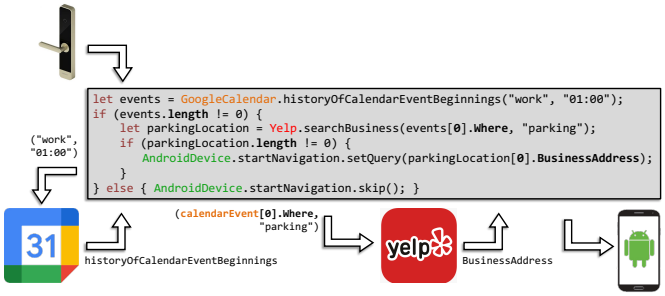

Figure 3: A parking space finding app.

In Figure 3, closing the door is detected by the smart sensor, which triggers the app. Then, the app checks whether a work meeting is going to start in the next hour. If so, it queries to find the nearest parking to where the upcoming meeting is held. If a suitable parking spot is found, the app sends the location of the parking to the user's navigator.

This example represents the practicality of multiple queries relying on one another, creating a chain of dependent input sources. In this app, both the meeting and parking locations contain privacy-sensitive information whereas the latter depends on the former.

The app only uses the location of the next meeting, if existing, and the location of the parking space, if found. The other entries and attributes in the two query arrays need to be shared by the push-all approach. The increased expressiveness of allowing queries to depend on any input data, from both trigger and query services, is a benefit of LazyTAP over IFTTT and minTAP in addition to allowing for precise data minimization in a setting with queries and nondeterminism. Dynamic minTAP fails to support queries and, hence, also the more expressive version of queries, while an extended version of static minTAP would mark all four attributes visible in the app code as necessary. LazyTAP precisely picks up only the attributes on-demand, tightly following the three possible executions of the app.

III. LazyTAP

LazyTAP delivers precise data minimization in the presence of queries, and is both app compatible with IFTTT and expressible as an extension to IFTTT. This section introduces the architecture of LazyTAP and highlights the changes to the trigger and query services as well as to the IFTTT runtime required to integrate LazyTAP.

An app in IFTTT consists of app configuration and filter code [32]. The app configuration specifies the services involved in the app: one trigger, zero or more queries, and one or more actions. Figure 4 illustrates the execution procedure for a schematic example. Upon trigger, IFTTT creates trigger data (T), performs queries and creates query data (Q), and

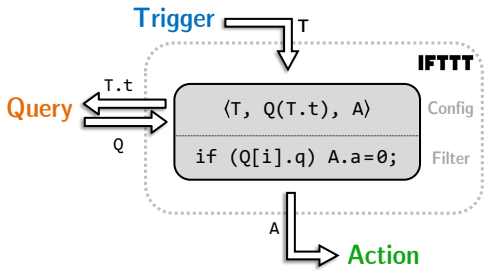

Figure 4: IFTTT architecture.

initializes action objects (A), all specified in the app configuration. Inputs to query services can be dependent only on the trigger data attributes (T.t) while action objects are configured by the data coming from the trigger and query services. The filter is a JavaScript code snippet regulating the app execution; it might overwrite the action object properties or skip any of the actions. In the figure, the value of one of the action object properties (A.a) implicitly depends on a query data attribute (Q[i].q). After executing the filter code, the resulting action objects are passed to the services to perform the corresponding actions.

A. Architecture of LazyTAP

The main difference between IFTTT and LazyTAP lies in how and when data is sent from the trigger and query services to the TAP. Instead of adopting the push-all approach of IFTTT, where all trigger data is pushed when an event occurs and all query data is fetched before running the app, LazyTAP develops a pull-on-demand paradigm.

Figure 5 illustrates the architecture of LazyTAP. To support the pull-on-demand approach, the data sent to the TAP by the trigger and the queries is replaced by access tokens that grant subsequent access to the data, allowing it to be fetched on a by-need basis. This change requires modifications to the trigger and query services in addition to the IFTTT runtime.

In the former case, we suggest the use of shims that adopt the original trigger and query services to the on-demand setting. It is an immediate choice as the input services are trusted by users and already obliged to meet the compatibility requirements of IFTTT [34]. Shims can be on the services, the straightforward approach which our prototype employs. A third party can provide the shim layer for the services that cannot be modified. Generally, while data minimization via on-demand computation is fundamental to LazyTAP, the use of shims is not, rather being subject to implementation alternatives.

In the latter case, we make use of two extensions to the IFTTT runtime, *remote objects* and *lazy queries*, in combination with deferred computation using thunking. Remote objects and lazy queries are used to give the filter code seamless access to the trigger and query data, while ensuring the on-demand approach and form the core of the app compatibility. The details of the shims, remote objects, and lazy queries are described below.

On trigger, LazyTAP creates a remote object representing the trigger data (T) backed by the given an access token

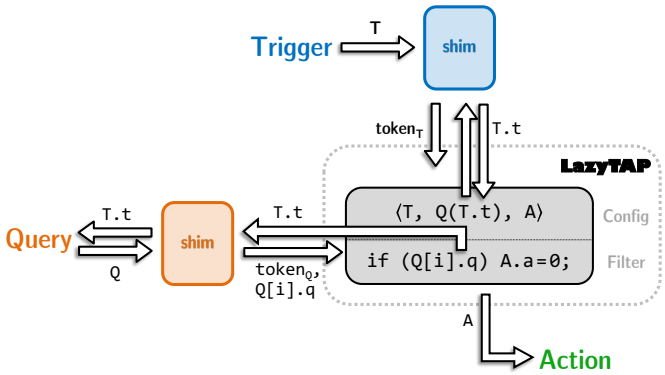

Figure 5: LazyTAP architecture.

(token$_T$), creates a remote object representing the query data (Q) backed by a lazy query, and initializes the action objects (A), all specified in the app configuration. Note that the computation of all query arguments and initial values of the action objects is deferred by thunking to prevent premature fetching of attributes.

The filter code is exactly the same code as in the IFTTT app execution. In the figure, based on a branch over a query data attribute (Q[i].q), the action object property (A.a) gets updated. The difference to the execution of the app in IFTTT is that the query is not performed until the query attribute is actually needed by the filter code. At this point, the trigger attribute (T.t) is fetched and the query is performed resulting in an access token (token$_Q$). In turn, this token is used to fetch the query data attribute. After executing the filter code, the resulting action objects are processed to perform any delayed computation and the result is passed to the services to perform the corresponding actions.

Trigger and query shims. To adopt the existing trigger and query services, we suggest the use of caching shims. Such shims can either execute on the trigger or query services, or be run on separate trusted services. For triggers, the shim service receives and caches the event data (T), and then generates an access token (token$_T$) that is sent to LazyTAP. The query shim works in a similar way. Instead of querying the service directly, LazyTAP queries the shim at the first access to a query data attribute (Q[i].q) during the app execution. To do so, LazyTAP prepares the query inputs by fetching the required attributes (T.t) from the trigger's shim and sends the query. The shim service then forwards the query to the actual query service and receives the result (Q). From this point, the shim service acts identically to the trigger's shim, sending an access token (token$_Q$) along with the requested attribute.

Remote objects and lazy queries. Ordinary IFTTT apps assume that the data from the trigger and queries is present as an object tree in the execution environment. For the sake of data minimization and to retain app compatibility, we introduce the notion of remote object and lazy query that provide seamless integration of the existing app into the LazyTAP fetch-on-demand setting.

Remote objects rely on JavaScript proxies, special objects

that allow for programmatic capture of any object interaction [19]. In particular, remote objects are proxies that intercept every read and write to them. A remote object is either associated with an access token and a base path that identifies the position of the remote object in the object tree or with a lazy query.

When reading a property on a remote object backed by an access token, one of two things can happen depending on whether the property has been accessed before or not. In the former case, the value is fetched from the cache and returned, while in the latter case, the access token, the base path, and the property name are used to fetch, cache, and return a new value. If the fetched property is a primitive value, it is used without further modification. Otherwise, if the fetched property is indicated to be an object, a new remote object is created, extending the base path with the property name. The caching mechanism prevents values from being fetched more than once.

When reading a property on a remote object backed by a lazy query, the query is first performed yielding an access token. This token together with an empty base path then replaces the lazy query and is used to perform the triggering and future property reads. Remote arrays extend remote objects to include the special property `length`.

Lazy queries are queries that are not initialized or performed until their first use. However, since queries can depend on trigger data, we need a way to avoid their creation causing premature reads of trigger data. For this reason, we allow thunking of query arguments. Thunking is a common way to encode delayed computation and is performed by wrapping a computation in a function. The thunked computation, e.g., the projection of a remote object, is delayed until the thunk is invoked. This way we can create lazy queries that simply store their thunked arguments until the first use. On the first use, the lazy queries evaluate the arguments, set up the query and return an access token associated with the query result.

The LazyTAP runtime. To develop a fully seamless runtime for app developers of IFTTT, we must change how the execution environment of the app is built. Essentially, we replace trigger data with the remote object establishing the connection to the trigger shim. In a similar way, for each query service, we replace the query data with the corresponding remote array object. We pass a thunked value as the query input to defer computing query inputs and the resulting object until the first read access of the query object. An important part of an app configuration is the specification of action object values. As mentioned earlier, the filter code might overwrite action objects or skip the execution of the action entirely, meaning that (parts of) the action objects defined in the app configuration are no longer needed. In case the initial values of the action objects rely on trigger or query data, the input data is at risk of being fetched prematurely. Thus, we thunk the action default values, making sure they are not assigned before running the filter code. After the execution of the filter code and before returning the action objects, we strictify and assign the default values to the action fields only if they have not been set earlier in the filter code. The post-app procedure guarantees that the preset values of the action objects are computed, possibly by fetching some data attributes from trigger or query services. Appendix A details these steps.

LazyTAP app compatibility and expressivity. It is important to note that the filter code remains untouched, as trigger and query objects are now proxy objects of the corresponding lazy services. Equally important is that the pre- and post-filter operations are naturally derived from the app configuration. Therefore, LazyTAP runs the original app using laziness to ensure that input data attributes are only fetched when they actually play a role in computing the final values of the action objects. LazyTAP is fully seamless to app developers and users as they notice no change in the execution behavior.

The use of lazy queries allows for greater freedom than what the current state of IFTTT does. For example, it is possible to make queries dependent on values from other queries, both directly by passing thunked query projections to queries and indirectly by creating different queries based on the values of other queries or trigger data. Thus, the programming model created by LazyTAP is more general and expressive than the one presented by IFTTT (see Section II-D). In fact, the essence of minimizing user data by on-demand computation can independently be applied to other TAP architectures.

B. On performance

In the proposed architecture, LazyTAP trades communication overhead for privacy. In the TAP setting, however, this is acceptable. For triggers, it rarely matters if the response is delayed by a few seconds. Network congestion and other factors already make services like IFTTT unreliable regarding response time. We report on the elapsed time for app executions under LazyTAP in our local setup and we discuss the effective factors in terms of performance in Section V. Further, some triggers provided by IFTTT are polling based. For such triggers, IFTTT offers a polling frequency of once every 15 minutes [42]. This shows that IFTTT is not meant to be a real-time service. This said, for apps that pull a lot of data the overhead can be significant. We envision that a static analysis that clusters data together to identify data that is connected in the sense that if the root is fetched so is the remaining data can significantly improve the overhead. In such cases, instead of fetching a dependency tree part by part, the entire subtree is transferred as a JSON-encoded string.

IV. Formalization

To reason about the correctness and precision of LazyTAP, we formalize the core of our approach. The focus of our modeling is the interaction between lazy computation, lazy queries, remote objects (including fetching and caching of values), and side effects. Since a full model of JavaScript is out of the scope of this paper, we abstract away from language-specific and implementation details. Yet we ensure that the core of the formal model remains in a one-to-one

$$
\begin{aligned}
e &::= v \mid x \mid e \oplus e \mid f(e) \mid e[e] \mid \{\} \mid T \mid Q(k, e) \mid A(m) \\
&\quad \mid () \Rightarrow e \\
i &::= x \mid i[e] \\
c &::= \mathit{skip} \mid i := e \mid \mathit{if}\ e\ \mathit{then}\ c\ \mathit{else}\ c \mid \mathit{while}\ e\ \mathit{do}\ c \mid c; c
\end{aligned}
$$

Figure 6: Language syntax.

correspondence with the core of the implementation in a semantic sense.

A. Syntax

We model the app code as a while language with objects, trigger data, queries, and actions. We assume the app configuration is given in the style of IFTTT apps, described in Section III. We present two semantics for the language: one strict semantics for IFTTT apps (Section IV-B) and one lazy semantics for the corresponding LazyTAP apps (Section IV-C), introducing the execution steps of the app code together with the given app configuration.

The syntax of the language is presented in Figure 6. The expressions, denoted by e, contain primitive values, variables, binary operators, function call of built-in functions, computed object projection, empty object creation, the dedicated syntax for accessing the trigger data T, setting up queries Q, and accessing actions A, as well as thunks $() \Rightarrow e$. Note that thunk expressions are *not* part of the strict syntax. Query setup and actions take an argument, $k \in \mathit{Query}$ and $m \in \mathit{Action}$, respectively, identifying the query or action service. To simplify the formal model and without loss of generality, we assume that the built-in functions and queries only take a single argument. Multiple arguments can be encoded as objects. For the same reason, methods are encoded by passing the object the method is invoked on as part of such argument object and arrays are encoded as integer indexed objects with a dedicated *length* property. The encodings are elaborated in Appendix B.

The statements, denoted by c, are *skip*, assignment to variables and properties, *if* statements, *while* statements, and sequences. Together expressions and statements model the core of IFTTT and LazyTAP apps.

As explained in Section III-A, LazyTAP converts trigger and query objects to remote objects and employs thunk expressions to defer computation, both formalized in the lazy semantics. Figure 7 shows a LazyTAP app in the while language, transformed from the corresponding IFTTT app, where the thunk constructs are added to the query setup expression. It is a snippet of the model of app B, introduced in Section II-B. The query object `calendar` is the result of the query to Google Calendar meeting events for today. The filter code sets the `Title` property of the `Slack` action object to the title of the meeting if taking place in office.

```
1 // query setup
2 calendar := () ⇒ Q(GoogleCalendar, () ⇒ today);
3 // filter code
4 if (calendar[Where] == office) then
5     A(Slack)[Title] := calendar[Title]
```

Figure 7: A snippet of the lazy model of app B.

B. Strict semantics

We present the strict semantics for IFTTT apps. Let the strict values $sv \in \mathit{SVal} ::= pv \mid r$ be primitive values $pv \in \mathit{PVal}$ and references $r \in \mathit{Ref}$. Let variable environments $E : \mathit{Var} \to \mathit{SVal}$ be mappings from variables to strict values, objects $o : \mathit{Prop} \to \mathit{SVal}$ be mappings from properties $p \in \mathit{Prop}$, and heaps $H : \mathit{Ref} \to \mathit{SObj}$ be maps from references to objects.

An app configuration $\Gamma = \langle t, q, a \rangle$ is a triple of the reference to the strict trigger object $t \in \mathit{Ref}$, a query function $q : (\mathit{Query} \times \mathit{String}) \to \mathit{String}$ that takes a query identifier and a string then returns a string representing the query result, and a mapping $a : \mathit{Action} \to \mathit{Ref}$ from action identifiers to references to the corresponding action objects. Queries taking and returning strings models that query services receive their arguments and return their results encoded as JSON objects. We assume two functions $\mathit{encJSON} : \mathit{SVal} \to \mathit{String}$ and $\mathit{decJSON} : \mathit{String} \times H \to (\mathit{Ref} \times H)$ that encode primitive values and decode string into object trees.

We present the strict semantics using two big-step evaluation relations, one for the expressions of the form $\Gamma \models (e, E, H) \Downarrow_s (sv, H)$, and one for statements of the form $\Gamma \models (c, E, H) \to_s (E, H)$. Evaluation of expressions computes an expression to a strict value, given the app configuration, a variable environment, and a heap. Statements are environment transformers mapping variable environments and heaps to (potentially) modified environments and heaps. Note that expression evaluation may modify the heap due to function calls and queries. The evaluation rules are mostly standard. For space reasons, we explain a selection of the non-standard rules, where Appendix D presents the complete set of rules.

Function calls. We assume a set of primitive functions that model the execution environment of JavaScript. Function calls are performed using the *apply* function that takes the function name, the argument, and a heap, then returns the result and a possibly modified heap. This allows functions to model method calls as well.

$$
\frac{\Gamma \models (e, E, H_1) \Downarrow_s (sv_1, H_2) \quad \mathit{apply}(f, sv_1, H_2) = (sv_2, H_3)}{\Gamma \models (f(e), E, H_1) \Downarrow_s (sv_2, H_3)} \quad \mathit{sevCall}
$$

Trigger evaluation. Trigger evaluation simply returns the trigger reference. Note that the trigger reference is given as part of the app configuration.

$$
\frac{}{\langle t, q, a \rangle \models (T, E, H) \Downarrow_s (t, H)} \quad \mathit{sevTrigger}
$$

Query evaluation. Queries are performed by first encoding the argument as a JSON string, performing the query, and then decoding the resulting string. Decoding the returned result may need creating an object tree, where the decoding function may modify the heap.

$$\frac{\langle t,q,a\rangle \models (e,E,H_1) \Downarrow_s (sv,H_2) \quad q(k,encJSON(sv)) = j \quad decJSON(j,H_2) = (r,H_3)}{\langle t,q,a\rangle \models (Q(k,e),E,H_1) \Downarrow_s (r,H_3)} \; sevQuery$$

Action evaluation. The action evaluation simply returns the action object associated with the action service.

$$\frac{a(m) = r}{\langle t,q,a\rangle \models (A(m),E,H) \Downarrow_s (r,H)} \; sevAction$$

C. Lazy semantics

We present the lazy semantics for LazyTAP apps, modeling the execution steps of the transformed runtime where trigger and query objects are remote objects (see Appendix A for information about the transformation).

In our implementation [2], there are separate classes for remote objects (RemoteObject) and arrays (RemoteArray) that use proxies to implement the fetching and caching in a seamless manner. Both rely on a class for lazy services (LazyService) that implements lazy queries and triggers. In the formalization, those classes are represented by dedicated syntax (instead of being encodable in the formalized language), where the semantics for the dedicated syntax captures the semantics of the implementation. Thus, remote projection in the lazy semantics models the execution of the remote objects and arrays, while fetching models the execution of the lazy service together with the strictification performed on the arguments of lazy queries when the corresponding remote object is projected.

Let the remote values $rv \in RVal ::= sv \mid r$ be strict values or remote references $r \in Ref_R \subset Ref$, i.e., references to remote objects, defined below. Let lazy values $lv \in LVal ::= rv \mid thunk(e)$ be remote values and lazy computations (thunks). We build a corresponding lazy execution environment as follows. Let lazy variable environments $E : Var \to LVal$ be maps from variables to lazy values, lazy objects $o : Prop \to LVal$ be maps from properties to lazy values, and lazy heaps $H : Ref \to LObj$ be maps from references to lazy objects. In addition, let remote heaps $R : Ref \to Fetcher \uplus (Query \times LVal)$ be mappings from remote references to either fetchers or lazy queries. Remote objects are modeled by references $r \in Ref_R$, where the lazy heap maps the reference to the cache objects and the remote heap maps the reference to either a lazy query or a fetcher.

Unlike the strict semantics for IFTTT apps, LazyTAP does not assume that the data from the trigger and queries is in the initial execution environment as a complete object tree. Instead, the remote trigger object is a fetcher and the remote query object is a lazy query. The fetcher (b, f_F) is a pair of a base path b and a fetcher function f_F, where the fetcher function is used to perform on-demand fetching of properties using the base and the property name. On the first interaction with a remote query object, the lazy query is performed and replaced by a fetcher. This is how trigger and query objects are treated similarly in LazyTAP. Note that remote objects are immutable, a key condition to guarantee the bijection relation between the strict and lazy execution environments.

A lazy app configuration $\Gamma = \langle t,q,a\rangle$ is a triple of a reference $t \in Ref$ to a remote trigger object, a query function $q : (Query \times String) \to Fetcher$ that returns fetchers instead of returning a string, and a mapping $a : Action \to Ref$ from action identifiers to references to the corresponding action objects. We assume two functions $encJSON : SVal \to String$ and $decJSON : String \to PVal \uplus \{\texttt{unit}\}$ that encode a query parameter as a string and decode the result. Note that the decoding now either returns a primitive value or an indication that the fetched property contains an object, rather than sending over the entire objects.

We present the lazy semantics using two big-step evaluation relations, one for expressions of the form $\Gamma \models (e,E,R,H) \Downarrow_l (lv,R,H)$ and one for statements of the form $\Gamma \models (c,E,R,H) \to_l (E,R,H)$. Evaluation of expressions computes an expression to a lazy value, given the lazy app configuration, a lazy variable environment, a lazy heap, and a remote heap. Statements are environment transformers mapping lazy variable environments and heaps to (potentially) modified variable environments and heaps. Note that expression evaluation may modify the heaps due to function calls and queries. For space reasons, we only explain a selection of the non-standard rules pertaining to projection of remote objects, query establishment, and thunk creation. Appendix D presents the complete set of rules.

Projection. The lazy semantics has both local objects and remote objects, reflected in the semantics for projection.

$$\frac{\Gamma \models (e_1,E,R_1,H_1) \Downarrow_l (r,R_2,H_2) \quad \Gamma \models (e_2,E,R_2,H_2) \Downarrow_l (p,R_3,H_3) \quad H_3(r) = o \quad o(p) = rv}{\Gamma \models (e_1[e_2],E,R_1,H_1) \Downarrow_l (rv,R_3,H_3)} \; levPrjLocal$$

$$\frac{\Gamma \models (e_1,E,R_1,H_1) \Downarrow_l (r,R_2,H_2) \quad \Gamma \models (e_2,E,R_2,H_2) \Downarrow_l (p,R_3,H_3) \quad RProject(\Gamma,r,p,E,R_3,H_3) = (rv,R_4,H_4)}{\Gamma \models (e_1[e_2],E,R_1,H_1) \Downarrow_l (rv,R_4,H_4)} \; levPrjRemote$$

Remote projection. Intuitively, remote projections are performed by first looking in the cache.

$$\frac{H(r) = o \quad o(p) = rv}{RProject(\Gamma,r,p,E,R,H) = (rv,R,H)} \; Cache$$

If the value has not been fetched yet, the fetcher is used to fetch the value before caching it. In the example shown in Figure 7, the projection access to `Where` and `Title` of the calendar query object calls the fetcher and caches the values.

$$\frac{R_1(r) = F \quad H_1(r) = o_1 \quad p \notin dom(o_1) \quad FetchDecode(F,p,R_1,H_1) = (rv,R_2,H_2) \quad o_2 = o_1[p \mapsto rv] \quad H_3 = H_2[r \mapsto o_2]}{RProject(\Gamma,r,p,E,R_1,H_1) = (rv,R_2,H_3)} \; Fetch$$

Fetching. Depending on whether the fetched value is a primitive value or an object, either the value is returned or a new remote object is created by extending the base of the fetcher.

$$\frac{f_F(b.p) = j \qquad \mathit{decJSON}(j) = pv \qquad F = (b, f_F)}{\mathit{FetchDecode}(F, p, R, H) = (pv, R, H)} \quad \mathit{fetchValue}$$

$$\frac{\begin{array}{c} f_F(b.p) = j \qquad \mathit{decJSON}(j) = \texttt{unit} \\ r \notin \mathit{dom}(H_1) \qquad r \notin \mathit{dom}(R_1) \\ H_2 = H_1[r \mapsto \{\}] \qquad R_2 = R_1[r \mapsto (b.p, f_F)] \\ F_1 = (b, f_F) \qquad F_2 = (b.p, f_F) \end{array}}{\mathit{FetchDecode}(F_1, p, R_1, H_1) = (r, R_2, H_2)} \quad \mathit{fetchObject}$$

In case the remote object is backed by a lazy query (e.g., Line 4 in Figure 7), the query must first be performed before using the resulting fetcher to fetch the value.

$$\frac{\begin{array}{c} R_1(r) = (k, lv) \\ \langle t, q, a\rangle \models (lv, E, R_1, H_1) \downarrow_s (rv, R_2, H_2) \\ q(k, \mathit{encJSON}(rv)) = F \qquad R_3 = R_2[r \mapsto F] \\ \mathit{Rproject}(\langle t, q, a\rangle, r, p, E, R_3, H_2) = (rv, R_4, H_3) \end{array}}{\mathit{RProject}(\langle t, q, a\rangle, r, p, E, R_1, H_1) = (rv, R_4, H_3)} \quad \mathit{Query}$$

Strictification. Since lazy queries may contain lazy values, we must strictify the argument before performing the query. Strictification only affects thunks that are performed. Other values are returned as is.

$$\frac{}{\Gamma \models (rv, E, R, H) \downarrow_s (rv, R, H)} \quad \mathit{RVal}$$

$$\frac{\Gamma \models (e, E, R_1, H_1) \Downarrow_l (rv, R_2, H_2)}{\Gamma \models (\mathit{thunk}(e), E, R_1, H_1) \downarrow_s (rv, R_2, H_2)} \quad \mathit{Thunk}$$

Queries and thunks. Lazy queries are not performed on the fly, but rather a new remote object backed by the lazy query is created (e.g., Line 2 in Figure 7).

$$\frac{\begin{array}{c} \Gamma \models (e, E, R_1, H_1) \Downarrow_l (lv, R_2, H_2) \\ \mathit{dom}(r) \notin R_2 \qquad \mathit{dom}(r) \notin H_2 \\ R_3 = R_2[r \mapsto (k, lv)] \qquad H_3 = H_2[r \mapsto \{\}] \end{array}}{\langle t, q, a\rangle \models (Q(k, e), E, R_1, H_1) \Downarrow_l (r, R_3, H_3)} \quad \mathit{levLQuery}$$

To allow for lazy queries, it is important that we can defer projection of remote objects until the need arises. This is possible with the help of thunking, which simply stores the thunked expression for later execution. There are limitations on how thunking is allowed to ensure correctness. Thunks may not be nested. Also, the free variables of a thunk must only refer to remote objects. The immutability of the remote objects ensures that the thunk evaluates to the same result regardless of when evaluation takes place.

$$\frac{}{\Gamma \models (() \Rightarrow e, E, R, H) \Downarrow_l (\mathit{thunk}(e), R, H)} \quad \mathit{levThunk}$$

D. Correctness and precision

We show the correctness of the lazy semantics by proving that the execution of a program in the strict and the lazy semantics coincide. To do so formally, we prove preservation of a *lazy-models-strict* relation. The relation encodes that a lazy environment models a strict environment in the sense that if the lazy environment is completed, i.e., if all the remote values are fetched, the result is isomorphic to the strict environment. Let $\beta \in (\mathit{Ref} \cup \mathit{Ref} \times \mathit{Fetcher}) \times \mathit{Ref}$ be an annotated bijection on references and remote references. While the definition of the lazy-models-strict relation $(\Gamma, E, H) \simeq_\beta (\Gamma, E, H)$ is rather technical, the intuition behind it is more direct.

Figure 8 illustrates the relation. The gray triangles depict local object graphs and the white triangles represent remote object trees. The lazy-models-strict relation is rooted in the variables, trigger, and actions, extending pointwise on the heaps. Local object graphs are equal up to isomorphism, which is enforced by the use of the bijection β. In the figure, $(p_1, p_2) \in \beta$ holds since they both reside in the variable x in the respective environment. In turn, all local references in β are then demanded to be pointwise equivalent. Due to the use of JSON to transfer trigger and query data, remote object graphs are proper trees. The idea is to express that a remote object tree properly models the corresponding local object graph. This is depicted in two ways in the figure to show that this may occur at any point in the environment or on the heap. In the figure below, the variable y contains a remote object on the lazy heap and the corresponding local object on the strict heap; thus it causes $((p_4, f_2), p_3) \in \beta$, where f_2 is the fetcher function for the remote object tree. Unlike local objects, we cannot simply extend the equivalence relation pointwise because the remote object tree may be partial. Instead, we rely on a notion of path equivalence between the remote object tree (defined by the fetcher), its partial cache tree, and the corresponding local object graph. This suffices due to the fact that remote objects are immutable and free of cycles. We refer to a pair of a lazy and a strict environment that is related via the lazy-models-strict relation as equivalent environments.

We prove correctness of LazyTAP in terms of two theorems: one stating that given equivalent environments, LazyTAP can

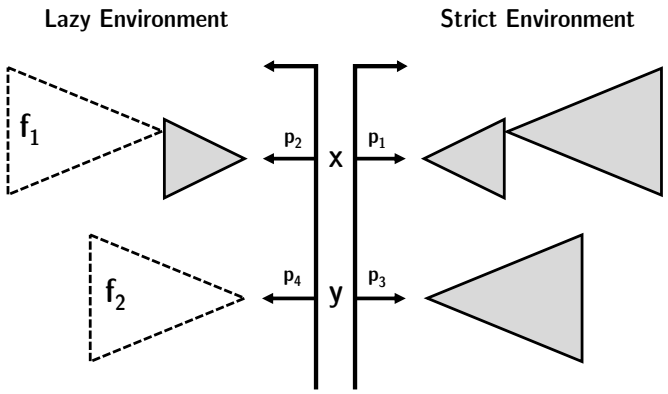

Figure 8: Lazy-strict isomorphism.

mimic any app execution of IFTTT (Theorem 1), and the other one that says LazyTAP does not add any execution that the strict semantics of IFTTT cannot perform (Theorem 2). Technically, we prove the theorems using two lemmas. The former (Simulation, in Appendix E) expresses that in equivalent environments, IFTTT is able to execute an app if and only if LazyTAP can. The latter (Preservation of the lazy-strict equivalence, in Appendix E) states that the resulting environments of such executions are indeed equivalent.

Relating strict execution to lazy execution relies on removing top-level thunking with the thunked expression, as thunks are not executable in the strict semantics. Only allowing top-level thunking simplifies the lazy semantics, which is in line with our restricted use of thunking in queries and actions. The removal of top-level thunks is performed by the *compileL2S*(c) function, described in Appendix C.

Theorem 1 (*LazyTAP apps model IFTTT apps*)**.** If the strict semantics is able to run, then so is the lazy semantics in every equivalent environment and the resulting environments are equivalent. Formally,

$$\begin{gathered}\forall c, c', \beta_1, \Gamma, E_1, R_1, H_1, \Gamma, E_1, H_1 E_2, H_2. \\ (\Gamma, E_1, R_1, H_1) \simeq_{\beta_1} (\Gamma, E_1, H_1) \wedge \\ c' = \mathit{compileL2S}(c) \wedge \\ \Gamma \models (c', E_1, H_1) \rightarrow_s (E_2, H_2) \Rightarrow \\ \exists \beta_2, E_2, R_2, H_2. \Gamma \models (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2) \wedge \\ \beta_1 \subseteq \beta_2 \wedge (\Gamma, E_2, R_2, H_2) \simeq_{\beta_2} (\Gamma, E_2, H_2).\end{gathered}$$

Proof. According to Simulation, the lazy semantics is able to follow any execution of the strict semantics in equivalent environments, and preservation gives that the resulting environments are indeed equivalent. □

Theorem 2 (*LazyTAP apps model only IFTTT apps*)**.** If the lazy semantics is able to run, then so is the strict semantics in every equivalent environment and the resulting environments are equivalent. Formally,

$$\begin{gathered}\forall c, c', \beta_1, \Gamma, E_1, R_1, H_1, \Gamma, E_1, H_1 E_2, R_2, H_2. \\ (\Gamma, E_1, R_1, H_1) \simeq_{\beta_1} (\Gamma, E_1, H_1) \wedge \\ c' = \mathit{compileL2S}(c) \wedge \\ \Gamma \models (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2) \Rightarrow \\ \exists \beta_2, E_2, H_2. \Gamma \models (c', E_1, H_1) \rightarrow_s (E_2, H_2) \wedge \\ \beta_1 \subseteq \beta_2 \wedge (\Gamma, E_2, R_2, H_2) \simeq_{\beta_2} (\Gamma, E_2, H_2).\end{gathered}$$

Proof. Based on Simulation, the strict semantics can follow any execution of the lazy semantics in equivalent environments. Preservation also states that the resulting environments are indeed equivalent. □

With the help of Correctness, we establish a general precision argument for all sound minTAP-style static and dynamic minimizers.

Theorem 3 (*Precision of LazyTAP*)**.** LazyTAP is at least as precise as any sound (static or dynamic) minimization technique; i.e., the resulting environment after executing LazyTAP will not contain more information than that produced by any preprocessing minimization technique.

Proof. Both static and dynamic minimization techniques result in minimized initial environments. Correctness of the static and dynamic minimization techniques gives that the app execution successfully maps the minimized initial environments to a final environment. The result follows from Theorem 1, as every strict environment has a lazy counterpart. □

Precision over static and dynamic minTAP follows immediately from Theorem 3 because both are sound minimization techniques based on data preprocessing.

Corollary 1 (*Precision of LazyTAP vs. static minTAP*)**.** LazyTAP is at least as precise as static minTAP.

Corollary 2 (*Precision of LazyTAP vs. dynamic minTAP*)**.** LazyTAP is at least as precise as dynamic minTAP (except for skipping executions of the apps without queries).

Note that the apps with queries are fundamentally out of the reach of dynamic minTAP. Static minTAP, however, overapproximates the required object properties while LazyTAP has access to the runtime values. In Figure 7, static minTAP puts the value of the query object property `calendar[Title]` in the minimized initial environment, due to the lack of execution prediction in advance. When the value of `calendar[Where]` is not office, applying the lazy semantic rules on the app yields a final environment of LazyTAP execution that does not contain unnecessary information of `calendar[Title]`.

Following from Theorem 3 and in line with the example above, LazyTAP attains a higher level of precision for the apps with queries for the executions that static minTAP is overly conservative; i.e., the minimized initial environment of minTAP might contain more information than the resulting environment produced under LazyTAP.

V. Evaluation

This section evaluates the minimization guarantees and performance of our implementation of LazyTAP. We have implemented LazyTAP in a setting based on IFTTT's architecture, including LazyTAP's runtime (see Appendix A) and app code as well as shimmed trigger and query services, explained in Section III-A. To ensure a fair comparison, we compare LazyTAP with IFTTT and a suggested extension on static minTAP [14] that conservatively identifies all the existing trigger and query attributes in the app code. We evaluate how LazyTAP minimizes trigger and query data transferred to the TAP on a collection of benchmarks. The study shows LazyTAP outperforms the extended minTAP by applying input-sensitive data minimization and preserves the behavior of app executions.

This section addresses the following research questions:

RQ1 How successful is LazyTAP in terms of data minimization for apps with various dependency patterns and code structures, including query chains (Section V-B)?

RQ2 How much minimization does LazyTAP achieve for the apps with filter code from the dataset [14] that make use of queries (Section V-C)?

RQ3 How much percentage does LazyTAP improve overall regarding attribute minimization compared to IFTTT and static analysis of minTAP (Section V-D)?

RQ4 How much overhead is imposed by LazyTAP compared to the original execution under IFTTT (Section V-E)?

A. Experimental setup

The benchmarks consist of two categories of IFTTT apps. The first category consists of six representative apps we developed, inspired by apps in IFTTT's store [27] to cover several classes of dependency patterns. The second category consists of actual apps with queries from the dataset of IFTTT apps including filter code [14]. Queries in apps is an emerging feature and also behind a paywall for users and app publishers, limiting the number of our benchmarks to what we present. Yet it is interesting that from the published apps with queries, we already observe a great variety of data dependencies in these apps, as represented by the benchmarks.

Out of 35 apps with queries in the dataset, we focus on the ones including filter code and privacy-sensitive trigger or query services, such as calendar events [22], list of personal tasks [23], and user-specific most-watched videos [49], resulting in 7 unique apps. Moreover, some insensitive queries like current weather [52] are used by several different apps in the dataset. Since data minimization also can significantly improve performance by not transferring attributes that are not used by the app execution, we include two privacy-insensitive yet popular apps in our experiments. Table II reports the trigger, query, and action services participating in each app.

We have implemented LazyTAP [2], readily deployable on AWS Lambda, using IFTTT's environment outlined in previous work [1]. Our performance evaluation was conducted on a macOS machine with a 2.4 GHz Quad-Core Intel Core i5 processor and 16 GB RAM.

Table III evaluates the minimization of trigger and query data attributes for the 15 apps, ranging from the representative cases to the extracted apps from the dataset. Each row reports the total number of trigger and query data attributes, the number of attributes after the static analysis of the extended minTAP, and the number of fetched attributes by LazyTAP per possible execution, respectively. The next two columns show how much data LazyTAP minimized on average over IFTTT and static minTAP. The last two columns report how many milliseconds it takes to run the app in our local IFTTT runtime and for each app execution in LazyTAP. We report the average elapsed time of 10 runs for each execution path. The reported latencies include the full execution time, from once the trigger token is received by the TAP to the point that all action outputs are populated, on a single machine to abstract away from the variability of network-related latency. The multiple numbers reported for LazyTAP (attributes and execution time) reflect all possible execution paths of the app. IFTTT apps are typically small snippets without complex constructs, thus we ensure full path coverage in our evaluation using synthetic data. Since the execution time for different paths of an app in IFTTT does not change noticeably, we report the average of all measured times. Dynamic minTAP is not present in the table due to the lack of support for apps with queries.

Table IV in Appendix F summarizes the description of the benchmarks by indicating the sensitivity of services, dependency relationships, and code patterns. For each app, it shows which services are sensitive, whether queries are dependent on trigger/query data or entirely independent, whether the skip commands depend on trigger/query data or time, and which data is present in the default values of action services, respectively. The last column specifies if the app has some unique features, such as nondeterministic query results, query chaining, or time-sensitive computation.

B. Dependency patterns (representative apps)

To answer RQ1 we investigate the advantage of LazyTAP runtime concerning minimization. We explain how a variety of representative apps with different types of dependency patterns and code structures execute under LazyTAP.

Inspired by an official app [25], app *A* connects Email to Slack. Depending on the sender of the incoming email (the attribute `From`), the Slack object gets different values; thus `From` is always accessed. If the sender is the supervisor, the Slack post requires three more attributes of Email, thus this path requires four attributes. If the email is a newsletter, the Slack post only obtains the email's timestamp, accessing two attributes in total for this path. Otherwise, the app skips the action entirely, touching only one attribute. IFTTT asks for all the seven Email attributes, no matter if they are necessary. The over-approximating analysis of static minTAP returns all of the five attributes visible in the source code. LazyTAP, however, only fetches the demanded attributes per execution. Dynamic minTAP, which is only applicable for this representative app without queries, refuses to send any trigger data for the skipping execution to the TAP.

App *B* is the motivating example in Section II-B and a modified instance of an IFTTT connection [29]. At a specific time of the day, the user's important calendar events happening in office are posted to Slack. The inputs to the sensitive query are constant values. The execution branches on the `Where` property of the first element of the query result. While IFTTT reads all the trigger attributes and the whole query array, static minTAP stipulates that the three present attributes in the code are needed. LazyTAP precisely fetches the accessed attributes for each execution and no more: one if the condition holds and three otherwise.

App *C* is a combination of the first two apps, including Email as another sensitive source of information. The query is whether a meeting with the email sender is in today's schedule. If not, the app skips; otherwise, the Slack message should contain the attributes `Subject`, `Body`, and `From` of Email, the

App Id	Trigger	Queries	Actions
A	Email.sendIftttAnEmail	-	Slack.postToChannel
B	DateAndTime.everyWeekdayAt	GoogleCalendar.historyOfEventFromSearchStarts	Slack.postToChannel
C	Email.sendIftttAnEmail	GoogleCalendar.historyOfEventFromSearchStarts	Slack.postToChannel
D	DateAndTime.everyWeekdayAt	Youtube.recentLikedVideos	Telegram.sendMessage
E	Smartlife.doorClosed	GoogleCalendar.historyOfCalendarEventBeginnings Yelp.searchBusiness	AndroidDevice.startNavigation
F	Email.sendIftttAnEmail	GoogleCalendar.historyOfEventFromSearchStarts Fitbit.historyOfDailyActivitySummaries	Slack.postToChannel Gmail.sendAnEmail
G	DateAndTime.everyWeekdayAt	Trakt.mostWatchedMovies	Twitter.postNewTweet
H	Location.enterRegionLocation	GoogleCalendar.searchEvents	WemoSwitch.attributeSocketOnDiscrete
I	Bouncie.fuelEcon	Finance.historyOfClosingPrices	Moretrees.plantTreeForSelf IfNotifications.sendNotification
J	Smartthings.switchedOnSmartthings	Trakt.recommendedMovies	IfNotifications.sendRichNotification
K	DateAndTime.everyWeekdayAt	Trakt.recommendedMovies Yelp.searchBusiness	Email.sendMeEmail
L	DateAndTime.everyWeekdayAt	GoogleTasks.listAllTasks	Slack.postToChannel
M	GoogleCalendar.newEventAdded	Giphy.historyOfRandomGifBasedOnKeyword	Slack.postToChannel
N	DateAndTime.everyDayAt	Weather.twoDayForecast	GoogleCalendar.quickAddEvent IfNotifications.sendRichNotification
O	Space.spaceStationOverheadSoonNasa	Weather.currentWeather	IfNotifications.sendNotification

Table II: LazyTAP benchmark services.

title of the first meeting, and the `length` of the calendar query. IFTTT always has access to the whole array of calendar events while static minTAP restricts the TAP's access to all of the five attributes. For the skipping execution, however, LazyTAP fetches only `From` and `length`.

App *D* is one of the apps with nondeterministic query results, similar to the motivating example in Section II-C. The app randomly suggests one of the recently liked YouTube videos of the user and posts it to Telegram. If there is no video (`length` is zero), the action is skipped. Otherwise, a random index is picked and the Telegram message concatenates `Title`, `Url`, and `Description` of the selected video. As the behavior of the app depends on a random value generated in the execution, static minTAP cannot foresee the index and returns the whole query array restricting each element to the three attributes (3*YT in Table I). In this case, LazyTAP outperforms static minTAP by far due to fetching only the four attributes with their actual values.

LazyTAP treats trigger and query services in a similar vein, allowing any kind of dependency between the trigger and the query services. App *E*, the example in Section II-D, is an illustrative app showing LazyTAP is able to handle apps with queries that can depend on each other, creating query chains, beyond what is possible with IFTTT or minTAP. The input to Yelp is the location from the calendar event, the first sensitive query. If there is no upcoming calendar event, the second query should not be performed at all. There are three different execution paths for this app: the action is skipped if there is no meeting (accessing `length` of `GoogleCalendar`), or when no parking area is close to the meeting's venue (also accessing `Where` of the first calendar event and `Yelp`'s `length`); otherwise the parking location is being sent to the navigator device (also accessing `BusinessAddress` of `Yelp`'s first element). Static minTAP marks all of the four attributes as necessary while LazyTAP only fetches the attributes of each execution path. IFTTT does not support query chaining.

A combination of app *C* and an existing IFTTT app [24] produces *F*, another genuine IFTTT app, where two query and two action services are involved. If the email is from the supervisor, the app notifies the user on Slack if there is a supervision meeting in the calendar. If user's personal trainer has sent an email, the app retrieves some attributes of Fitbit's daily activity and auto-replies by sending the collected information. If the email's sender is neither of the two, both actions are skipped. Although Fitbit does not ask for the email's sender as a query input, performing each of the queries implicitly depends on the trigger data. In lieu of transferring all the trigger data and both query arrays to the TAP, static minTAP aggregates all of the ten attributes in both branches. Again, the on-demand approach of LazyTAP brings accuracy by minimizing the data attributes per execution: six for the first branch and five for the second one.

C. Dataset analysis (apps with queries)

To answer RQ2 we study the IFTTT apps with filter code from the dataset [14] that include queries.

App *G* [39] slices the query array of user's most-watched movies up to ten elements and iterates over the subarray to tweet `MovieTitle` and `MovieYear` of each movie. With the assumption of applying a static analyzer that understands how array slicing works, static minTAP can imaginably predict which attributes of every element of the subarray are used. LazyTAP, on the other hand, has access to values in the execution, including `length`, assuring precision in data minimization by execution.

App *H* is one of the apps introduced in IFTTT's documentation [26]. The action is to heat up a location if the user arrives there on the same day, according to the calendar. The two

App Id	Attributes					Execution Time (ms)	
	Total (IFTTT)	Static minTAP	LazyTAP	Over IFTTT(%)	Over minTAP(%)	IFTTT	LazyTAP
A	7	5	1, 2, 4	66.7	53.3	9	127, 232, 438
B	3 + (7 * GC)	3	1, 3	99.4	33.3	130	234, 438
C	8 + (7 * GC)	5	2, 5	99.0	30.0	129	339, 650
D	3 + (5 * YT)	1 + (3 * YT)	1, 4	99.6	33.3	131	235, 552
E	4 + (7 * GC) + (7 * YL)	4	1, 3, 4	99.6	65.0	N/A	237, 544, 649
F	9 + (7 * GC) + (12 * FI)	10	1, 2, 5, 6	99.0	96.7	236	129, 340, 654, 752
G	3 + (5 * TK)	1 + min(TK, 10) * 2	1 + min(TK, 10) * 2	92.6	0.0	132	233, 443, 652, ..., 2431
H	4 + (10 * GC)	2	2	99.6	0.0	133	336
I	4 + (5 * FN)	4	3	98.8	25.0	130	443
J	3 + (7 * TK)	1 + TK	1, 2, 3, 4	99.3	90.4	132	231, 335, 442, 550
K	4 + (7 * TK) + (7 * YL)	3 + TK + YL	4	99.4	92.5	234	650
L	3 + (7 * GT)	1 + (3 * GT)	1, ..., 1 + (3 * GT)	78.5	0.0	135	233, 335, 445, 548, ..., 15286
M	9 + (6 * GP)	4	2, 4	99.0	25.0	132	233, 544
N	2 + (16 * WL)	4	4	99.5	0.0	131	549
O	6 + (20 * WL)	5	3, 5	99.6	20.0	130	446, 651

Table III: LazyTAP benchmark evaluation. Number of attributes includes length of query arrays. Abbreviations: GoogleCalendarLength (GC), YoutubeLength (YT), YelpLength (YL), FitbitLength (FI), TraktLength (TK), FinanceLength (FN), GoogleTasksLength (GT), GiphyLength (GP), WeatherLength (WL).

attributes `OccurredAt` and `searchEvents[0].Start` are always accessed in all executions, thus static minTAP and LazyTAP minimize the same number of attributes.

App *I* [36] uses regex and `parseFloat` methods on the `FuelEcon` and `Price` attributes. Source code analysis of minTAP shows the default value of the action object needs two more attributes whereas the action object is always overwritten and only one more attribute is accessed.

Apps *J* [37], simplified in Section II-C, randomly recommends three movies based on the user's preferences, which can be potentially repeated. Similarly, App *K* [40] suggests a random movie and a random restaurant, influenced by user's data. Akin to app *D*, static minTAP fails at accurately minimizing apps with nondeterministic query attributes and returns all movie titles (and restaurant names for app *K*). LazyTAP executes the app and fetches the randomly picked index of the queries, preserving the app's behavior including possibly duplicate values for app *J*.

App *L* [28] iterates over user's personal tasks, posting `Title` and `Note` of each if the `Due` date is today. Depending on the due dates, which cannot be predictable for a static analyzer, LazyTAP outperforms static minTAP with a smaller or larger margin. Only when the due date of all tasks is today, the number of attributes in LazyTAP and static minTAP coincides.

The filter code of App *M* [31] depends only on the sensitive trigger data, meaning that performing the query is unnecessary if the action is skipped. LazyTAP communicates with the query service only if needed, saving two attributes more than static minTAP for the skipped executions.

The last two apps, *N* [26] and *O* [30], exemplify how much redundant data is being sent to IFTTT for some insensitive queries like weather. Each element of the query array contains more than 16 attributes while less than five are actually accessed by the app, showing that LazyTAP uses network bandwidth efficiently thanks to on-demand data minimization.

D. Minimization

To answer RQ3 we measure how much LazyTAP on average improves data minimization over IFTTT and static minTAP. Table III includes two columns to report how much data on average has been minimized using LazyTAP compared to IFTTT and static minTAP. LazyTAP has multiple numbers of fetched attributes with respect to the path taken by the app execution. To compute the accurate weighted average of the overall time overhead, the statistical distribution of paths taken should be given. Because of missing this information, we simply consider the average of the reported numbers for a given app under LazyTAP. To calculate the percentage of improvement in terms of data minimization, the parametric numbers of the attributes in the table for IFTTT and static minTAP should also be quantified. Due to the lack of statistical information on the length of query results for the various services in question, we assign an average value for the size of query arrays. The default maximum limit for any query response is 50 events [34]; thus, we replace the query length variables with 25 to have an unbiased average value. Note that we assume IFTTT only has access to the most recent trigger event, unlike the real-world setup where trigger services are expected to send the 50 most recent events [34], regardless of whether or not they have already been sent. According to the calculated numbers for our benchmarks in the two columns, LazyTAP does not fetch 95% of users' data otherwise transferred to IFTTT and improves minimization by 38% over minTAP.

E. Performance

To answer RQ4 we discuss the time and cost overhead of LazyTAP versus IFTTT. The reported numbers in Table III indicate that the number of query services has a direct impact on IFTTT's execution time for an app, in our local deployment. In all cases except for four cases, the execution time is

approximately 130ms, because of one trigger and one query service in the apps. Apps F and K consume more time in IFTTT because of the additional query service. App A has no queries, computing the action object immediately.

According to the numbers for executions under LazyTAP, the time overhead correlates with the number of fetched attributes. For example, fetching four attributes in an app takes between 438ms to 650ms, varying based on the data size. Our benchmark shows on average 150ms and no longer than 250ms spent to fetch an attribute. The worst case is an execution of app L that takes 15 seconds to fetch 151 unique attributes, which is well within 15 minutes of IFTTT's guarantee to poll the trigger service [42]. The optimization techniques mentioned in Section III-B enhance the performance of LazyTAP even further.

Note that the connections to the shim services can be kept alive during the app execution, meaning that even if the number of requests grows, the number of connections remains only one per service. In a pricing model of serverless deployments charging per request, however, the cost might increase. It would be onerous to estimate the cost impact in a general setting because both factors of the volume of requests and computation time matter.

VI. Related work

Recent surveys [9], [11], [3], [13] overview the state-of-the-art on the security and privacy of TAPs.

Privacy on TAPs. Previous work shows that overprivileged access to trigger/action APIs [20] opens up for harvesting private information [53] and enables malicious rule makers to exploit TAP's privileges [10], [1]. Privacy-sensitive endpoints on IFTTT available via triggers and queries include various personal data, including location and health data [41]. This raises privacy concerns tackled by our work.

DTAP [20] focuses on the integrity of apps under a malicious TAP [20]. DTAP relies on extending the OAuth protocol with so-called XTokens to express fine-grained privileges and requires a trusted client to configure the apps. In contrast, LazyTAP addresses data privacy. OTAP [16] achieves data privacy with respect to TAPs by encryption and padding techniques. This approach can protect data in transit, but it does not allow computing on the data by TAPs (by, e.g., filter code), a key feature of TAPs. In contrast, LazyTAP focuses on data minimization and offers fully-fledged support for filter code on the TAP. LazyTAP can be extended with encryption of attributes in the style of OTAP.

eTAP [15] also targets privacy with respect to a malicious TAP, and, in contrast to OTAP, supports computation on the TAP. This is achieved by garbled circuits for app execution. While it provides strong confidentiality and integrity guarantees, it only supports a limited subset of features in filter code and incurs higher overhead.

Filter-and-Fuzz [53] explores how events from a smart home can be sanitized to ensure that IFTTT does not learn more information than necessary. It relies on textual analysis to identify unnecessary events. LazyTAP can benefit from hiding statistical patterns of sensitive events by composing them with the Fuzzing part of Filter-and-Fuzz.

minTAP [14] is subject to detailed comparisons throughout the paper, summarized in Table I. Compared to minTAP's focus on helping trigger services to sanitize their data before is passed to a potentially ill-intended TAP, LazyTAP helps the TAP itself to obtain fine-grained data minimization by on-demand computation and the benefits it entails in terms of support for queries and nondeterminism. Since minTAP and LazyTAP use IFTTT as a starting point for experimentation, their prototypes inherit some of IFTTT's elements of architecture. At a conceptual level, however, LazyTAP's design is different from minTAP because of pull-on-demand computations via lazy proxy objects to load necessary remote data behind the scenes. Architecturally, LazyTAP's prototype requires the integration of the proxying into IFTTT's runtime, whereas minTAP requires a separate trusted client.

Secure hardware. Recent efforts leverage secure hardware for protecting users' data from TAPs. Hardware-based trusted execution environments (TEEs) enable computing over the trigger data on the TAP while preserving confidentiality [55], [47]. Besides requiring hardware changes to the TAP backends, current TEEs suffer from fundamental security design issues [50], [44], [12].

App security studies. This line of work analyzes the semantics of trigger-action rules to determine conflicts or security policy violations (e.g., a door opens when it should not) [48], [17], [51], [3]. While important, this work is orthogonal to LazyTAP as it deals with the semantics of rule sets, rather than the data privacy issues that arise from the fundamental design shortcomings of TAPs.

Language-based data minimization and minimum exposure. Data minimization is a principle restricting data collection to "what is necessary in relation to the purposes for which they are processed" [21]. Antignac et al. [8] formalize the notions of monolithic and distributed data minimization. Pinisetty et al. [46] utilize testing techniques for data minimization, but leave synthesizing minimizers as future work. Compared to this line of work, we develop practical data minimization techniques that focus on the attributes used by programs.

Anciaux et al. [5], [7], [6] focus on the case of collecting forms (like tax forms) for governments. They consider the number of inputs to withhold for the privacy of the applicants and discuss data-dependent minimum exposure. However, the computational model is that of assertions on particular shapes of formulas that represent form collection logic, making their algorithmic solutions less applicable to scenarios of general programs. By contrast, our approach naturally extends the language-based approach to data minimization, which applies to arbitrary (runs of) programs.

VII. Conclusion

We have presented LazyTAP, a fine-grained on-demand paradigm for trigger-action applications that warrants input-sensitive data minimization by design. In contrast to the

previous approaches, LazyTAP supports both multiple triggers/queries and nondeterministic/randomized behaviors of the apps. We leverage laziness and proxy objects to develop a novel architecture for on-demand computation for third-party JavaScript apps, loading necessary remote data behind the scenes. This achieves full backward compatibility for app developers. We formally establish the correctness of LazyTAP and its minimization properties with respect to both IFTTT and minTAP. We implement and evaluate LazyTAP on app benchmarks showing that on average LazyTAP improves minimization by 95% over IFTTT and by 38% over minTAP, while incurring a tolerable performance overhead.

Acknowledgments. Thanks are due to Musard Balliu, Earlence Fernandes, Sandro Stucki, and the anonymous reviewers for their valuable feedback. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF), and the Swedish Research Council (VR).

REFERENCES

[1] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security*, 2021.

[2] M. M. Ahmadpanah, D. Hedin, and A. Sabelfeld. LazyTAP implementation and benchmarks. https://www.cse.chalmers.se/research/group/security/lazytap/, 2023.

[3] M. Alhanahnah, C. Stevens, and H. Bagheri. Scalable analysis of interaction threats in IoT systems. In *ISSTA*, 2020.

[4] Amazon. AWS Lambda. https://aws.amazon.com/lambda/, 2023.

[5] N. Anciaux, W. Bezza, B. Nguyen, and M. Vazirgiannis. Minexp-card: limiting data collection using a smart card. In *EDBT*, 2013.

[6] N. Anciaux, D. Boutara, B. Nguyen, and M. Vazirgiannis. Limiting data exposure in multi-label classification processes. *Fundam. Informaticae*, 2015.

[7] N. Anciaux, B. Nguyen, and M. Vazirgiannis. Limiting data collection in application forms: A real-case application of a founding privacy principle. In *PST*, 2012.

[8] T. Antignac, D. Sands, and G. Schneider. Data minimisation: A language-based approach. In *SEC*, 2017.

[9] M. Balliu, I. Bastys, and A. Sabelfeld. Securing IoT Apps. *IEEE Security & Privacy Magazine*, 2019.

[10] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *CCS*, 2018.

[11] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. D. McDaniel. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Computing Surveys*, 2019.

[12] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *EuroS&P*, 2019.

[13] X. Chen, X. Zhang, M. Elliot, X. Wang, and F. Wang. Fix the leaking tap: A survey of trigger-action programming (TAP) security issues, detection techniques and solutions. *Comput. Secur.*, 2022.

[14] Y. Chen, M. Alhanahnah, A. Sabelfeld, R. Chatterjee, and E. Fernandes. Practical data access minimization in trigger-action platforms. In *USENIX Security*, 2022.

[15] Y. Chen, A. R. Chowdhury, R. Wang, A. Sabelfeld, R. Chatterjee, and E. Fernandes. Data Privacy in Trigger-Action Systems. In *S&P*, 2021.

[16] Y.-H. Chiang, H.-C. Hsiao, C.-M. Yu, and T. H.-J. Kim. On the Privacy Risks of Compromised Trigger-Action Platforms. In L. Chen, N. Li, K. Liang, and S. Schneider, editors, *ESORICS*, 2020.

[17] C. Cobb, M. Surbatovich, A. Kawakami, M. Sharif, L. Bauer, A. Das, and L. Jia. How risky are real users' IFTTT applets? In *SOUPS*, 2020.

[18] California Privacy Rights Act (CPRA). https://oag.ca.gov/privacy/, 2020.

[19] ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. https://262.ecma-international.org/6.0, 2023.

[20] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized action integrity for trigger-action iot platforms. In *NDSS*, 2018.

[21] General Data Protection Regulation (GDPR). Art. 5 Principles relating to processing of personal data. https://gdpr-info.eu/art-5-gdpr/, 2018.

[22] GoogleCalendar. Search events of a calendar. https://ifttt.com/google_calendar/queries/search_events, 2023.

[23] GoogleTasks. List all tasks in a list. https://ifttt.com/google_tasks/queries/list_all_tasks, 2023.

[24] Daily Fitbit activity summary emailed to me. https://ifttt.com/applets/rPh7NHe6, 2023.

[25] Email a message to a Slack channel. https://ifttt.com/applets/EJVR4sz8, 2023.

[26] Example applets using queries and filter code. https://help.ifttt.com/hc/en-us/articles/360053657913-Example-Applets-using-queries-and-filter-code, 2023.

[27] IFTTT. Explore Applets. https://ifttt.com/explore/applets, 2023.

[28] Get a daily recap on Slack of all my Google Tasks due today. https://ifttt.com/applets/YG5HSLvK, 2023.

[29] Get a morning reminder about your first meeting daily. https://ifttt.com/connections/WHQ7AjWP, 2023.

[30] Get a notification when the ISS passes over your house but only if it is clear skies and after dark. https://ifttt.com/applets/VDdNBmiE, 2023.

[31] Get Slack notifications for new calendar events without an agenda. https://ifttt.com/applets/xvyUBQsh, 2023.

[32] IFTTT. IFTTT: Creating Applets. https://platform.ifttt.com/docs/applets, 2023.

[33] IFTTT: If This Then That. https://ifttt.com, 2023.

[34] IFTTT. IFTTT: Service API requirements. https://platform.ifttt.com/docs/api_reference, 2023.

[35] IFTTT. IFTTT's Glossary: Query. https://platform.ifttt.com/docs/glossary}query, 2023.

[36] Plant trees when your car trips have less than ideal fuel economy. https://ifttt.com/applets/iqZPNUtR, 2023.

[37] Saturday movie night recommendations with Samsung SmartThings and Trackt. https://ifttt.com/applets/jUy5if7H, 2023.

[38] IFTTT. The art of the query. https://ifttt.com/developer_blog/the-art-of-the-query, 2023.

[39] Tweet your most watched movies every week! https://ifttt.com/applets/AxJSC34d, 2023.

[40] Weekly date night email. https://ifttt.com/applets/MRm9VBxG, 2023.

[41] S. Kalantari, D. Hughes, and B. De Deckerd. Listing the ingredients for ifttt recipes. In *TrustCom*, 2022.

[42] X. Mi, F. Qian, Y. Zhang, and X. Wang. An empirical characterization of ifttt: ecosystem, usage, and performance. In *Internet Measurement*, 2017.

[43] Microsoft Power Automate. https://powerautomate.microsoft.com, 2023.

[44] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.

[45] A. Pfitzmann and M. Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf, 2010.

[46] S. Pinisetty, T. Antignac, D. Sands, and G. Schneider. Monitoring data minimisation. *CoRR*, abs/1801.02484, 2018.

[47] S. Schoettler, A. Thompson, R. Gopalakrishna, and T. Gupta. Walnut: A low-trust trigger-action platform. https://arxiv.org/pdf/2009.12447.pdf, 2020.

[48] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *WWW*, 2017.

[49] Trakt. List my most watched movies. https://ifttt.com/trakt/queries/most_watched_movies, 2023.

[50] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX*, 2018.

[51] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter. Charting the attack surface of trigger-action IoT platforms. In *CCS*, 2019.

[52] WeatherUnderground. Get the current weather. https://ifttt.com/weather/queries/current_weather, 2023.

[53] R. Xu, Q. Zeng, L. Zhu, H. Chi, X. Du, and M. Guizani. Privacy leakage in smart homes and its mitigation: IFTTT as a case study. *IEEE Access*, 2019.
[54] Zapier. https://zapier.com, 2023.
[55] I. Zavalyshyn, N. Santos, R. Sadre, and A. Legay. My House, My Rules: A Private-by-Design Smart Home Platform. In *EAI MobiQuitous*, 2020.

Appendix A: Transformation of runtime

Figure 9 exemplifies how IFTTT's runtime is transformed into LazyTAP. The process proceeds as follows:

Step 1. Replace `triggerData`, the data existing in the body of the trigger service communication, with the remote object establishing the connection to the lazy service of the trigger; i.e., `RemoteObject.Create(new LazyService(triggerUrl))` (Line 2-3).

Step 2. For each query service, replace the object `queryService(queryUrl, queryInput)` with the remote object establishing the connection to the lazy service of the query; i.e., `RemoteArray.Create(new LazyService(queryUrl, () => queryInput))` (Line 6-7).

Step 3. For each action service, for each action field `field_i`, thunk the value by prepending `() =>` to the value `defaultValue_i` (Line 11-12).

Step 4. Omit assigning the default values to the action fields before the filter code by removing the `Object.assign` invocations for each action service (Line 14).

Step 5. For each action service, include the postapp code snippet that strictifies and assigns the default values to the action fields only if they have not been set in the filter code.

Appendix B: Encoding of methods and arrays

Methods can be encoded by using an object to carry the arguments and an object the method is invoked on as follows.

```
1 // compilation of method call o.f(a)
2 x := { }
3 x["this"] := o
4 x[0] := a;
5 f(x)
```

Arrays can be encoded as number indexed objects with a special `length` property. From a modeling perspective, it is assumed any methods impacting array's size modify the `length` property accordingly.

```
1 // compilation of arrays x = [a_0, ... , a_n]
2 x = { }
3 x[0] = a_0
4 ...
5 x[n] = a_n
6 x["length"] = n + 1
```

Appendix C: Lazy-to-strict compilation

```
1 compileL2S (Exp e) =
2   case () ⇒ e : e
3   case Q(k, e) : Q(k, compileL2S(e))
4   default : e
```

```
1 compileL2S (Cmd c) =
2   case i := e : i := compileL2S(e)
3   case c1; c2 : compileL2S(c1); compileL2S(c2)
4   default : c
```

Appendix D: Semantic rules

The strict evaluation rules for expressions are found in Figure 10, and the strict execution rules for statements are found in Figure 11. The lazy evaluation is introduced in Figure 12, the lazy execution is presented in Figure 13, and the supporting relations are found in Figure 14.

Appendix E: Correctness

Figure 15 presents the rules for the equivalence relation.

Lemma 1 (*Preservation of equivalence of statements*)**.** Execution maintains equivalence. Formally,

$$\begin{gathered}\forall c, c', \beta_1, \Gamma, E_1, R_1, H_1, \Gamma, E_1, H_1, E_2, R_2, H_2, E_2, H_2.\\ (\Gamma, E_1, R_1, H_1) \simeq_{\beta_1} (\Gamma, E_1, H_1) \wedge\\ \Gamma \models (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2) \wedge\\ c' = compileL2S(c) \wedge\\ \Gamma \models (c', E_1, H_1) \rightarrow_s (E_2, H_2) \Rightarrow\\ \exists \beta_2.\, \beta_1 \subseteq \beta_2 \wedge (\Gamma, E_2, R_2, H_2) \simeq_{\beta_2} (\Gamma, E_2, H_2).\end{gathered}$$

Proof. By induction over the height of execution tree using preservation of equivalence of expressions. □

Lemma 2 (*Simulation of statements*)**.** The strict semantics executes successfully if and only if the lazy semantics executes successfully. Formally,

$$\begin{gathered}\forall c, c', \beta_1, \Gamma, E_1, R_1, H_1, \Gamma, E_1, H_1.\\ (\Gamma, E_1, R_1, H_1) \simeq_{\beta_1} (\Gamma, E_1, H_1) \wedge\\ c' = compileL2S(c)\\ \Rightarrow \Big(\big(\forall E_2, R_2, H_2, .\, \Gamma \models (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2) \Rightarrow\\ \exists E_2, H_2.\, \Gamma \models (c', E_1, H_1) \rightarrow_s (E_2, H_2)\big) \wedge\\ \big(\forall E_2, H_2.\, \Gamma \models (c', E_1, H_1) \rightarrow_s (E_2, H_2) \Rightarrow\\ \exists E_2, R_2, H_2.\, \Gamma \models (c, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2)\big)\Big).\end{gathered}$$

Proof. By induction over the height of execution tree. The proof is standard and makes use of preservation of equivalence of statements and expressions as well as the proof of simulation of expressions. □

Appendix F: LazyTAP benchmark

Table IV describes the dependency patterns of the apps in the benchmark.

```
var TriggerService = {
-  triggerName: triggerData
+  triggerName: RemoteObject.Create(new LazyService(triggerUrl))
}
var QueryService = {
-  queryName: queryService(queryUrl, queryInput)
+  queryName: RemoteArray.Create(new LazyService(queryUrl, () => queryInput))
}
var ActionService = { actionName: { skipped: false } }
var actionDefaultValues = {  ActionService: {  actionName: {
-  field_i: defaultValue_i
+  field_i: () => defaultValue_i
}  }  }
-  Object.assign(ActionService.actionName, actionDefaultValues["ActionService"]["actionName"])
ActionService.actionName.setField_i = function(value) { if (!this.skipped) this.field_i = value }
ActionService.actionName.skip = function() { this.skipped = true }
//end of app configuration
filterCode
+  if (!ActionService.actionName.skipped) {
+    var actionfields = actionDefaultValues["ActionService"]["actionName"]
+    for (const field in actionfields) {
+      if (!ActionService.actionName.hasOwnProperty(field)) {
+        ActionService.actionName[field] = actionfields[field]() //strictify the thunk value
+  }  }  }
return ActionService
```

Figure 9: IFTTT-to-LazyTAP transformation of runtime.

Category	App Id	Sensitive Services	Query Depending On	Skip Based On	Preset Action Values	Other Features
Representative apps	*A*	T	-	T	T	-
	B	Q	I	-	-	-
	C	T, Q	T	Q	-	-
	D	Q	I	Q	-	Nondeterministic query results (`Math.random`)
	E	T, Q_1, Q_2	Q_1: I, Q_2: Q_1	Q_1, Q_2	Q_2	Query chain; Conditional second query
	F	T, Q_1, Q_2	Q_1: T, Q_2: T	T, Q_1, Q_2	A_1: -, A_2: T	Queries on trigger-dependent branches; Two actions
Apps from dataset	*G*	Q	I	-	Q	Array `slice`; `forEach` loop
	H	T, Q	I	T, Q	-	`Date` object from `moment`
	I	T	I	-	T	String methods using regex and `parseFloat`; Two actions
	J	T, Q	I	M	-	Nondeterministic query results (`Math.random`); `currentUserTime`
	K	Q_1	Q_1: I, Q_2: I	-	T	Nondeterministic query results (`Math.random`)
	L	Q	I	Q	Q	`currentUserTime` and `moment` object; `forEach` loop
	M	T	I	T	T, Q	String methods using regex; `forEach` loop
	N	-	I	-	A_1: -, A_2: -	String methods; Two actions
	O	-	I	Q, M	T	`currentUserTime`; String methods

Table IV: LazyTAP benchmark description of dependency patterns and app structures. T: Trigger, Q: Query, I: Independent; M: Time (moment.js). The app with id *E* is not a genuine IFTTT app due to the lack of support for query chaining.

$$\frac{}{\Gamma \models (pv, E, H) \Downarrow_s (pv, H)}\ sevVal \qquad \frac{E(x) = sv}{\Gamma \models (x, E, H) \Downarrow_s (sv, H)}\ sevVar \qquad \frac{\Gamma \models (e_1, E, H_1) \Downarrow_s (pv_1, H_2) \qquad \Gamma \models (e_2, E, H_2) \Downarrow_s (pv_2, H3)}{\Gamma \models (e_1 \oplus e_2, E, H_1) \Downarrow_s (pv_1 \oplus pv_2, H_3)}\ sevOPlus$$

$$\frac{\Gamma \models (e, E, H_1) \Downarrow_s (sv_1, H_2) \quad apply(f, sv_1, H_2) = (sv_2, H_3)}{\Gamma \models (f(e), E, H_1) \Downarrow_s (sv_2, H_3)}\ sevCall \qquad \frac{\Gamma \models (e_1, E, H_1) \Downarrow_s (r, H_2) \quad \Gamma \models (e_2, E, H_2) \Downarrow_s (p, H_3) \quad H_3(r) = o \quad o(p) = sv}{\Gamma \models (e_1[e_2], E, H_1) \Downarrow_s (sv, H_3)}\ sevPrj \qquad \frac{r \notin dom(H_1) \quad H_2 = H_1[r \mapsto \{\}]}{\Gamma \models (\{\}, E, H_1) \Downarrow_s (r, H_2)}\ sevNew$$

$$\frac{}{\langle t, q, a \rangle \models (T, E, H) \Downarrow_s (t, H)}\ sevTrigger \qquad \frac{\langle t, q, a \rangle \models (e, E, H_1) \Downarrow_s (sv, H_2) \quad q(k, encJSON(sv)) = j \quad decJSON(j, H_2) = (r, H_3)}{\langle t, q, a \rangle \models (Q(k, e), E, H_1) \Downarrow_s (r, H_3)}\ sevQuery \qquad \frac{a(m) = r}{\langle t, q, a \rangle \models (A(m), E, H) \Downarrow_s (r, H)}\ sevAction$$

Figure 10: Strict evaluation.

$$\frac{}{\Gamma \models (skip, E, H) \rightarrow_s (E, H)}\ seSkip \qquad \frac{\Gamma \models (e, E_1, H_1) \Downarrow_s (sv, H_2) \quad E_2 = E_1[x \mapsto sv]}{\Gamma \models (x := e, E_1, H_1) \rightarrow_s (E_2, H_2)}\ seAsn \qquad \frac{\Gamma \models (c_1, E_1, H_1) \rightarrow_s^n (E_2, H_2) \quad \Gamma \models (c_2, E_2, H_2) \rightarrow_s^{n'} (E_3, H_3)}{\Gamma \models (c_1; c_2, E_1, H_1) \rightarrow_s^{n+n'+1} (E_3, H_3)}\ seSeq$$

$$\frac{\Gamma \models (i, E, H_1) \Downarrow_s (r, H_2) \quad \Gamma \models (e_1, E, H_2) \Downarrow_s (p, H_3) \quad \Gamma \models (e_2, E, H_3) \Downarrow_s (sv, H_4) \quad H_2(r) = o_1 \quad o_2 = o_1[p \mapsto sv] \quad H_5 = H_4[r \mapsto o_2]}{\Gamma \models (i[e_1] := e_2, E, H_1) \rightarrow_s (E, H_5)}\ seAsnPrj$$

$$\frac{\Gamma \models (e, E_1, H_1) \Downarrow_s (b, H_2) \quad \Gamma \models (c_{bool}, E_1, H_2) \rightarrow_s^n (E_2, H_3)}{\Gamma \models (\text{if } e \text{ then } c_{true} \text{ else } c_{false}, E_1, H_1) \rightarrow_s^{n+1} (E_2, H_3)}\ seIf \qquad \frac{\Gamma \models (e, E, H_1) \Downarrow_s (false, H_2)}{\Gamma \models (\text{while } e \text{ do } c, E, H_1) \rightarrow_s (E, H_2)}\ seWhile\text{-}false$$

$$\frac{\Gamma \models (e, E_1, H_1) \Downarrow_s (true, H_2) \quad \Gamma \models (c; \text{while } e \text{ do } c, E_1, H_2) \rightarrow_s^n (E_2, H_3)}{\Gamma \models (\text{while } e \text{ do } c, E_1, H_1) \rightarrow_s^{n+1} (E_2, H_3)}\ seWhile\text{-}true$$

Figure 11: Strict execution.

$$\frac{}{\Gamma \models (pv, E, R, H) \Downarrow_l (pv, R, H)}\ levVal \qquad \frac{E(x) = lv}{\Gamma \models (x, E, R, H) \Downarrow_l (lv, R, H)}\ levVar \qquad \frac{r \notin dom(H_1) \quad H_2 = H_1[r \mapsto \{\}]}{\Gamma \models (\{\}, E, R, H_1) \Downarrow_l (r, R, H_2)}\ levNew$$

$$\frac{\Gamma \models (e_1, E, R_1, H_1) \Downarrow_l (pv_1, R_2, H_2) \quad \Gamma \models (e_2, E, R_2, H_2) \Downarrow_l (pv_2, R_3, H_3)}{\Gamma \models (e_1 \oplus e_2, E, R_1, H_1) \Downarrow_l (pv_1 \oplus pv_2, R_3, H_3)}\ levOPlus \qquad \frac{\Gamma \models (e, E, R_1, H_1) \Downarrow_l (rv_1, R_2, H_2) \quad apply(f, rv_1, R_2, H_2) = (rv_2, R_3, H_3)}{\Gamma \models (f(e), E, R_1, H_1) \Downarrow_l (rv_2, R_3, H_3)}\ levFCall$$

$$\frac{\Gamma \models (e_1, E, R_1, H_1) \Downarrow_l (r, R_2, H_2) \quad \Gamma \models (e_2, E, R_2, H_2) \Downarrow_l (p, R_3, H_3) \quad H_3(r) = o \quad o(p) = rv}{\Gamma \models (e_1[e_2], E, R_1, H_1) \Downarrow_l (rv, R_3, H_3)}\ levPrjLocal \qquad \frac{\Gamma \models (e_1, E, R_1, H_1) \Downarrow_l (r, R_2, H_2) \quad \Gamma \models (e_2, E, R_2, H_2) \Downarrow_l (p, R_3, H_3) \quad RProject(\Gamma, r, p, E, R_3, H_3) = (rv, R_4, H_4)}{\Gamma \models (e_1[e_2], E, R_1, H_1) \Downarrow_l (rv, R_4, H_4)}\ levPrjRemote$$

$$\frac{}{\langle t, q, a \rangle \models (T, E, R, H) \Downarrow_l (t, R, H)}\ levTrigger$$

$$\frac{\Gamma \models (e, E, R_1, H_1) \Downarrow_l (lv, R_2, H_2) \quad dom(r) \notin R_2 \quad dom(r) \notin H_2 \quad R_3 = R_2[r \mapsto (k, lv)] \quad H_3 = H_2[r \mapsto \{\}]}{\langle t, q, a \rangle \models (Q(k, e), E, R_1, H_1) \Downarrow_l (r, R_3, H_3)}\ levLQuery$$

$$\frac{a(m) = r}{\langle t, q, a \rangle \models (A(m), E, R, H) \Downarrow_l (r, R, H)}\ levAction \qquad \frac{}{\Gamma \models (() \Rightarrow e, E, R, H) \Downarrow_l (thunk(e), R, H)}\ levThunk$$

Figure 12: Lazy evaluation.

$$\frac{}{\Gamma \models (skip, E, R, H) \rightarrow_l (E, R, H)}\ leSkip \qquad \frac{\Gamma \models (e, E_1, R_1, H_1) \Downarrow_l (lv, R_2, H_2) \quad E_2 = E_1[x \mapsto lv]}{\Gamma \models (x := e, E_1, R_1, H_1) \rightarrow_l (E_2, R_2, H_2)}\ leAsn$$

$$\frac{\Gamma \models (i, E, R_1, H_1) \Downarrow_l (r, R_2, H_2) \quad \Gamma \models (e_1, E, R_2, H_2) \Downarrow_l (p, R_3, H_3) \quad \Gamma \models (e_2, E, R_3, H_3) \Downarrow_l (rv, R_4, H_4) \quad H_2(r) = o_1 \quad o_2 = o_1[p \mapsto rv] \quad H_5 = H_4[r \mapsto o_2]}{\Gamma \models (i[e_1] := e_2, E, R_1, H_1) \rightarrow_l (E, R_4, H_5)}\ leAsnPrj$$

$$\frac{\Gamma \models (e, E_1, R_1, H_1) \Downarrow_l (bool, R_2, H_2) \quad \Gamma \models (c_{bool}, E_1, R_2, H_2) \rightarrow_l^n (E_2, R_3, H_3)}{\Gamma \models (\text{if } e \text{ then } c_{true} \text{ else } c_{false}, E_1, R_1, H_1) \rightarrow_l^{n+1} (E_2, R_3, H_3)}\ leIf$$

$$\frac{\Gamma \models (e, E_1, R_1, H_1) \Downarrow_l (true, R_2, H_2) \quad \Gamma \models (c; \text{while } e \text{ do } c, E_1, R_2, H_2) \rightarrow_l^n (E_2, R_3, H_3)}{\Gamma \models (\text{while } e \text{ do } c, E_1, R_1, H_1) \rightarrow_l^{n+1} (E_2, R_3, H_3)}\ leWhile\text{-}true \qquad \frac{\Gamma \models (e, E, R_1, H_1) \Downarrow_l (false, R_2, H_2)}{\Gamma \models (\text{while } e \text{ do } c, E, R_1, H_1) \rightarrow_l (E, R_2, H_2)}\ leWhile\text{-}false$$

$$\frac{\Gamma \models (c_1, E_1, R_1, H_1) \rightarrow_l^n (E_2, R_2, H_2) \quad \Gamma \models (c_2, E_2, R_2, H_2) \rightarrow_l^{n'} (E_3, R_3, H_3)}{\Gamma \models (c_1; c_2, E_1, R_1, H_1) \rightarrow_l^{n+n'+1} (E_3, R_3, H_3)}\ leSeq$$

Figure 13: Lazy execution.

$$\frac{f_F(b.p) = j \qquad decJSON(j) = pv \qquad F = (b, f_F)}{FetchDecode(F, p, R, H) = (pv, R, H)}\ fetchValue \qquad \frac{f_F(b.p) = j \qquad decJSON(j) = \texttt{unit} \qquad r \notin dom(H_1) \qquad r \notin dom(R_1) \qquad H_2 = H_1[r \mapsto \{\}] \qquad R_2 = R_1[r \mapsto (b.p, f_F)] \qquad F_1 = (b, f_F) \qquad F_2 = (b.p, f_F)}{FetchDecode(F_1, p, R_1, H_1) = (r, R_2, H_2)}\ fetchObject$$

$$\frac{H(r) = o \qquad o(p) = rv}{RProject(\Gamma, r, p, E, R, H) = (rv, R, H)}\ Cache \qquad \frac{R_1(r) = F \qquad H_1(r) = o_1 \qquad p \notin dom(o_1) \qquad FetchDecode(F, p, R_1, H_1) = (rv, R_2, H_2) \qquad o_2 = o_1[p \mapsto rv] \qquad H_3 = H_2[r \mapsto o_2]}{RProject(\Gamma, r, p, E, R_1, H_1) = (rv, R_2, H_3)}\ Fetch$$

$$\frac{\langle t, q, a\rangle \models (lv, E, R_1, H_1) \downarrow_s (rv, R_2, H_2) \qquad R_1(r) = (k, lv) \qquad q(k, encJSON(rv)) = F \qquad R_3 = R_2[r \mapsto F] \qquad Rproject(\langle t, q, a\rangle, r, p, E, R_3, H_2) = (rv, R_4, H_3)}{RProject(\langle t, q, a\rangle, r, p, E, R_1, H_1) = (rv, R_4, H_3)}\ Query$$

$$\frac{}{\Gamma \models (rv, E, R, H) \downarrow_s (rv, R, H)}\ RVal \qquad \frac{\Gamma \models (e, E, R_1, H_1) \Downarrow_l (rv, R_2, H_2)}{\Gamma \models (() \Rightarrow e, E, R_1, H_1) \downarrow_s (rv, R_2, H_2)}\ Thunk$$

Figure 14: FetchDecode, RProject, Strictify.

$$\frac{}{\Gamma, E, R, H, \Gamma, E, H \models pv \simeq_\beta pv}\ modelPrim \qquad \frac{(r_1, r_2) \in \beta}{\Gamma, E, R, H, \Gamma, E, H \models r_1 \simeq_\beta r_2}\ modelRef$$

$$\frac{R(r_1) = F \qquad ((r_1, F), r_2) \in \beta}{\Gamma, E, R, H, \Gamma, E, H \models r_1 \simeq_\beta r_2}\ modelRemote \qquad \frac{R(r_1) = (k, lv) \qquad \langle t, q, a\rangle \models (lv, E, R_1, H_1) \downarrow_s (rv, R_2, H_2) \qquad q(k, encJSON(rv)) = F \qquad ((r_1, F), r_2) \in \beta}{\langle t, q, a\rangle, E, R_1, H_1, \Gamma, E, H \models r_1 \simeq_\beta r_2}\ modelLQuery$$

$$\frac{\forall lv, R_2, H_2.\ \Gamma \models (e, E, R_1, H_1) \Downarrow_l (lv, R_2, H_2) \implies \exists \beta_2.\big(\beta_1 \subseteq \beta_2 \wedge (\Gamma, E, R_2, H_2) \simeq_{\beta_2} (\Gamma, E, H_2) \wedge \Gamma, E, R_2, H_2, \Gamma, E, H_2 \models lv \simeq_{\beta_2} sv\big)}{\Gamma, E, R_1, H_1, \Gamma, E, H_2 \models thunk(e) \simeq_{\beta_1} sv}\ modelLThunk$$

$$\frac{}{\Gamma, E, R, H, \Gamma, E, H \models pv \simeq_\beta^F pv}\ modelFetcherPrim \qquad \frac{H(r) = o \qquad \Gamma, E, R, H, \Gamma, E, H \models (\{\}, F) \simeq_\beta o}{\Gamma, E, R, H, \Gamma, E, H \models \texttt{unit} \simeq_\beta^F r}\ modelFetcherObject$$

$$\frac{dom(o) = dom(o) \qquad \big(\forall p.\ o(p) = rv \wedge o(p) = sv \implies \Gamma, E, R, H, \Gamma, E, H \models rv \simeq_\beta sv\big)}{\Gamma, E, R, H, \Gamma, E, H \models o \simeq_\beta o}\ objectModels$$

$$\frac{\begin{array}{c}\forall p.\ p \in dom(o) \implies p \in dom(o) \\ \forall p.\ (b.p) \in dom(f_F) \iff p \in dom(o) \\ \big(\forall p.\ o(p) = rv \wedge o(p) = sv \implies \Gamma, E, R, H, \Gamma, E, H \models rv \simeq_\beta sv\big) \\ \big(\forall p.\ o(p) = sv \wedge f_F(b.p) = j \wedge decJSON(j) = pv \implies \Gamma, E, R, H, \Gamma, E, H \models pv \simeq_\beta^{((b,p),f_F)} sv\big)\end{array}}{\Gamma, E, R, H, \Gamma, E, H \models (o, (b, f_F)) \simeq_\beta o}\ remoteModels \qquad \frac{\begin{array}{c}dom(E) = dom(E) \\ \big(\forall x.\ E(x) = lv \wedge E(x) = sv \implies \Gamma, E, R, H, \Gamma, E, H \models lv \simeq_\beta sv\big)\end{array}}{\Gamma, R, H, \Gamma, H \models E \simeq_\beta E}\ modelVarEnv$$

$$\frac{\begin{array}{c}\forall r_1, r_2.\ (r_1, r_2) \in \beta \implies r_1 \in dom(H) \wedge r_2 \in dom(H) \\ \forall r_1, r_2, F.\ ((r_1, F), r_2) \in \beta \implies r_1 \in dom(H) \wedge r_2 \in dom(H) \big(\forall r_1, r_2.\ (r_1, r_2) \in \beta \wedge H(r_1) = o \wedge H(r_2) = o \implies \Gamma, E, R, H, \Gamma, E, H \models o \simeq_\beta o\big) \\ \big(((r_1, F), r_2) \in \beta \wedge H(r_1) = o \wedge H(r_2) = o \implies \Gamma, E, R, H, \Gamma, E, H \models (o, F) \simeq_\beta o\big)\end{array}}{\Gamma, E, R, \Gamma, E \models H \simeq_\beta H}\ modelHeap$$

$$\frac{\begin{array}{c}dom(a) = dom(a) \\ \big(\forall m.\ a(m) = r_1 \wedge a(m) = r_2 \implies \langle t, q, a\rangle, E, R, H, \langle t, q, a\rangle, E, H \models r_1 \simeq_\beta r_2\big)\end{array}}{\langle t, q, a\rangle, E, R, H, \langle t, q, a\rangle, E, H \models a \simeq_\beta a}\ modelAction \qquad \frac{\begin{array}{c}\langle t, q, a\rangle, E, R, H, \langle t, q, a\rangle, E, H \models t \simeq_\beta t \\ \langle t, q, a\rangle, R, H, \langle t, q, a\rangle, H \models E \simeq_\beta E \\ \langle t, q, a\rangle, E, R, \langle t, q, a\rangle, E \models H \simeq_\beta H \\ \langle t, q, a\rangle, E, R, H, \langle t, q, a\rangle, E, H \models a \simeq_\beta a\end{array}}{(\langle t, q, a\rangle, E, R, H) \simeq_\beta (\langle t, q, a\rangle, E, H)}\ modelEnv$$

$$\frac{\begin{array}{c}\Big(\forall \beta_1, t, a, E, R_1, H_1, t, a, E, H_1, lv, sv, k, r_1. \\ \big((\langle t, q, a\rangle, E, R_1, H_1) \simeq_{\beta_1} (\langle t, q, a\rangle, E, H_1) \wedge \langle t, q, a\rangle, E, R_1, H_1, \langle t, q, a\rangle, E, H_1 \models lv \simeq_{\beta_1} sv \wedge \\ r_1 \notin dom(R_1) \wedge r_1 \notin dom(H_1) \wedge H_2 = H_1[r_1 \mapsto \{\}] \wedge R_2 = R_1[r_1 \mapsto (k, lv)] \wedge q(k, encJSON(sv)) = j \wedge decJSON(j, H_1) = (r_2, H_2)\big) \\ \implies \exists \beta_2.\ \big(\beta_1 \subseteq \beta_2 \wedge (\langle t, q, a\rangle, E, R_2, H_2) \simeq_{\beta_2} (\langle t, q, a\rangle, E, H_2) \wedge \langle t, q, a\rangle, E, R_2, H_2, \langle t, q, a\rangle, E, H_2 \models r_1 \simeq_{\beta_2} r_2\big)\Big)\end{array}}{q \simeq q}\ modelQueries$$

Figure 15: Strict-lazy equivalence of environments.