Data Privacy in Trigger-Action Systems

Yunang Chen, Amrita Roy Chowdhury, Ruizhe Wang,
Andrei Sabelfeld†, Rahul Chatterjee, Earlence Fernandes

University of Wisconsin–Madison † Chalmers University of Technology

***Abstract*—Trigger-action platforms (TAPs) allow users to connect independent web-based or IoT services to achieve useful automation. They provide a simple interface that helps end-users create trigger-compute-action rules that pass data between disparate Internet services. Unfortunately, TAPs introduce a large-scale security risk: if they are compromised, attackers will gain access to sensitive data for millions of users. To avoid this risk, we propose eTAP, a privacy-enhancing trigger-action platform that executes trigger-compute-action rules without accessing users' private data in plaintext or learning anything about the results of the computation. We use garbled circuits as a primitive, and leverage the unique structure of trigger-compute-action rules to make them practical. We formally state and prove the security guarantees of our protocols. We prototyped eTAP, which supports the most commonly used operations on popular commercial TAPs like IFTTT and Zapier. Specifically, it supports Boolean, arithmetic, and string operations on private trigger data and can run 100% of the top-500 rules of IFTTT users and 93.4% of all publicly-available rules on Zapier. Based on ten existing rules that exercise a wide variety of operations, we show that eTAP has a modest performance impact: on average rule execution latency increases by 70 ms (55%) and throughput reduces by 59%.**

I. Introduction

Trigger-action platforms (TAPs), such as IFTTT [8], Zapier [11], and Microsoft Power Automate [4] are web-based systems that enable users to stitch together their cyber-physical and digital resources (e.g., IoT devices, GMail, Instagram, Slack) to achieve useful automation. TAPs provide a simple *trigger-compute-action* paradigm and an easy-to-use interface to program automation rules.

For example, using their smartphone, a user can setup a rule that checks if an email contains the word "confidential" and, if so, sends an SMS with the subject line and the sender's address to a pre-specified number (Fig. 1). Instead of an SMS, the rule could also blink a smart light whenever a matching email arrives. To execute this rule on a TAP, when an email arrives (*trigger*), the mail service (*trigger service*) sends the email to the TAP that runs the string search (*computation*), which then contacts an SMS gateway or a smart bulb service (*action service*) with required information to perform the *action*. We refer to the combination of trigger/action services and the TAP as a trigger-action system — a key ingredient for fulfilling the promise of the IoT [49]. They provide a layer of abstraction that enables trigger and action services to develop APIs independently without worrying about compatibility with each other.

These benefits unfortunately come at the high price of private data disclosure to the TAPs. Even the simple rule discussed above reveals the user's private emails to the TAP.

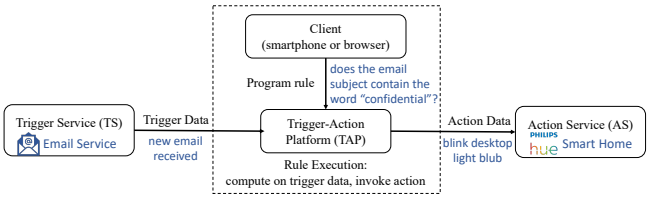

Fig. 1: Overview of current trigger-action systems. The dataflow for the example rule is illustrated in blue color: "IF I receive an email containing the word 'confidential', THEN blink my desktop smart light."

As the TAP is the center of communication between triggers and actions, it can launch *person-in-the-middle* attacks by invisibly collecting private information on all of its users, similar to what has already been happening on centralized ride-hailing platforms [34], [47]. Due to the highly compatible nature of TAPs, this data includes location, voice commands, fitness data, pictures, files, etc. [38] and is limited only by the variety of online services of users (e.g., IFTTT supports 600 services [37]). Commercial TAPs do not provide any technical protections for user data. For example, IFTTT's terms of use explicitly state that they collect personal data from third parties, and may pass it to other third parties, partners, or any company that might acquire IFTTT [38].

Furthermore, because TAPs are widely-used centralized web services (e.g., IFTTT has more than 20 million users [40]), they are attractive targets for attackers. Breaches of cloud services are commonplace [3], [60], [63], [72]. Attackers sometimes even have continued access to the compromised service for days, and even weeks before getting detected [25], [26], [59]. A similar breach will have disastrous consequences for TAP users. Such privacy risks might discourage users as well as trigger/action services from using TAPs. Indeed GMail, due to security and privacy concerns, pulled back some of its APIs from IFTTT [61].

In this paper, we introduce eTAP, an encrypted trigger-action platform that executes user rules without accessing the underlying user data in plaintext. Thus, eTAP provides confidentiality even when the attacker fully controls the TAP. Although this problem fits in the general framework of secure function evaluation (SFE) [74], [75], building a functional and secure trigger-action platform with good performance requires overcoming several challenges.

First, we desire confidentiality of user's data and authenticity of computation when the TAP is compromised and acts maliciously. While there are protocols for SFE that provide

security even if some parties act maliciously [31], [48], these constructions are not yet practical [48], [64]. Second, using off-the-shelf protocols for SFE will require invasive changes to the architecture of trigger-action systems that break the independence between trigger and action services, making them less useful. Third, running arbitrary computations on the TAP using SFE will be inefficient.

We leverage the unique structure and threat model of trigger-action systems to overcome these challenges. At a high-level, we create a trusted generator of garbled circuits (GCs). This allows eTAP to use semi-honest implementations of SFE coupled with a few efficient extensions, which we contribute with security proofs, to achieve security against a fully malicious circuit evaluator.

In our setting, the user's smartphone, a standard component in TAP design, plays the role of a trusted circuit generator that periodically generates and transmits garbled circuits to the untrusted TAP. The trigger service garbles sensitive data when it is available and calls the TAP, which then executes the circuit and contacts the action service with the (garbled) results. The action service performs security checks and then executes the action. We assume that the user's phone is fully trusted, while TAP is malicious. An attacker interested in compromising a large number of users is more likely to try compromising the TAP than the user's phone. To maintain the same level of trust as current TAPs provide, we treat the trigger and action services as semi-honest — they follow the protocol but can be inquisitive — and they should not learn any new private information that they do not learn in the current setting.

To overcome the challenge concerning the efficiency of arbitrary computations, we perform an analysis of the types of computations in popular commercial trigger-action platforms. We show that the computations supported by TAPs are stateless and use Boolean, arithmetic, or string operations. Most GC libraries support Boolean and arithmetic operations natively, but none support string operations out of the box. Existing work contributes oblivious deterministic finite automata that can match regular expressions [51]. However, it does not support substring extraction and replacements — a common operation in trigger-action systems. We therefore introduce a novel approach to efficiently encode a subset of fixed-length string operations as Boolean circuits. We then use the standard GC approach to evaluate them securely on the TAP. Our approach also has the advantage of unifying all the formal security properties of eTAP rather than having a separate set of proofs for string operations. eTAP can compute 93.4% of all rules published on Zapier that require computation and 100% of the 500 most-used rules on IFTTT. (Of course, eTAP supports all rules that do not require any computation.)

We formally prove the security of eTAP in the presence of a malicious TAP (Section A). We show that the malicious TAP can execute user rules without learning the private data or tampering with the result of computation. eTAP also provides mutual secrecy between the trigger and action services.

eTAP is a clean-slate approach to building trigger-action systems and lays a foundation for securing the data they handle. However, it does require some changes to current systems. First, the trigger/action services need to understand our protocols. We provide simple shims that they can use to upgrade their functionality while maintaining their independence and RESTful nature. Second, the user's client device takes on a more prominent role because it generates garbled circuits. As efficient circuits cannot be reused in general, the client has to periodically generate and transmit these circuits to the TAP. We estimate that this process has a modest impact: the trusted client is expected to transfer 61.7 MB of data per day for an average user. This is equivalent to the data consumed by uploading a one-minute of Full-HD video.

The paper offers the following contributions:

- We design eTAP, the first trigger-action platform that can execute trigger-compute-action rules (Boolean, arithmetic, fixed-length string) without accessing the underlying trigger data in plaintext.
- We outline ideal security expectations of a privacy-sensitive trigger-action system, and formally prove that eTAP meets those security properties.
- We implement and evaluate eTAP. It can support 93.4% of function-dependent rules used in Zapier and 100% of the 500 most-used rules in IFTTT. We show that most functions can be evaluated in the order of milliseconds with about 2x computational cost. Code is available at https://github.com/EarlMadSec/etap.

II. Background

We discuss background information on trigger-action systems and the cryptographic primitives that we use.

A. Trigger-Action Systems

Trigger-action systems allow stitching together disparate online services using a trigger-compute-action paradigm to automate different tasks. There are three main components of the system: trigger services (TSs), action services (ASs), and a trigger-action platform (TAP). We also explicitly mention another computing component: the user's client device that they use to interface with the trigger-action system. Fig. 1 shows the interactions between different components.

Trigger and action services are online services for IoT or web apps. There are a plethora of such services such as Instagram, Slack, GMail, Amazon Alexa, Samsung SmartThings, and many others. These services rely on REST APIs to send and receive data, and each service may support several APIs to provide different functionalities. They typically support the OAuth protocol [58], which is used to delegate authorization. With OAuth tokens, a third party, such as a TAP, can access APIs and execute trigger-compute-action rules.

Commercial TAPs are compatible with hundreds of trigger and action services, allowing each trigger or action service to focus on building their own REST APIs without worrying about compatibility with each other. Third-parties own a large

majority of these services that integrate with IFTTT (e.g., LG, Samsung, Google).[1]

Additionally, modern TAPs also allow performing non-trivial computation over the trigger data. The ability to modify the trigger data provides great flexibility for TAPs to achieve compatibility between trigger and action services (e.g., two calendar apps that use different date formats). The TAP also uses operations to decide whether or not it should send a message to the action service (e.g., does the email contain the word "confidential"). TAPs serve as a computation and communication hub. Zapier has explicitly supported computation on trigger data from the very beginning [6], [7]. IFTTT has recently started to expose its computing interface to end-users [39]. Thus, trigger-action systems are evolving to be *trigger-compute-action* systems. We use these two terms interchangeably throughout the paper.

Users interface with trigger-action system through a client device, typically a smartphone. The user programs rules by selecting a trigger service, then specifying a computation on that data using a library of functions, and finally selecting an action to be run on the action service. As noted before, the user also authorizes the TAP to access their online services using the client device.

Privacy and authenticity risks in current TAPs. Commercial TAPs operate on sensitive trigger data of millions of users, making them an attractive target for attackers. If the TAP is compromised, the attacker gains the privilege of the TAP — unfettered access to user data and resources. The types of data are limited only by the set of rules that users create and the end-point services that the TAP supports. Commercial systems like IFTTT support approximately 600 services currently [37]. The sensitive information from these services can be emails (our earlier example), data files, health information, voice commands, images, etc.

Fernandes et al. [30] first noted this problem with TAPs, and discussed a more appropriate threat model where TAP can act maliciously. Under this model, they addressed a sub-problem: preventing a compromised TAP from misusing overprivileged OAuth tokens. Their work adds integrity to the rules, but it does not allow any computation over the trigger data.

By contrast, we target modern TAPs that allow computation over the trigger data. Beyond integrity, we also aim to protect the *privacy* of that data. Our work provides a way for TAPs to compute on sensitive data without seeing the plaintext, despite arbitrarily deviating from the protocol. We believe such privacy risks might be preventing trigger-action systems from achieving their true potential. Furthermore, we provide computational integrity as well, thus subsuming prior work [30].

B. Cryptographic Primitives

Symmetric-key encryption scheme. Let $\mathcal{E} = (\mathsf{K}, \mathsf{E}, \mathsf{D})$ be a semantically secure encryption scheme. The key generation function $\mathsf{K}(1^\kappa)$ generates a κ-bit uniformly random key k; the randomized encryption scheme E takes a message $x \in \mathcal{X}$ and the generated key k as input and outputs a cipher text $ct \leftarrow\!\!\$\ \mathsf{E}(k, x)$; and the deterministic decryption function takes a cipher text and the key k as input and outputs a message, $x \leftarrow \mathsf{D}(k, ct)$, or $\perp$ (if decryption fails).

We use an authenticated encryption scheme [18] that achieves the IND-CCA security guarantee. This ensures both the privacy and authenticity of plaintext.

Garbled circuits (GCs). This is a cryptographic technique for secure function evaluation (SFE) [18], [76]. Following Bellare et al.'s [17] notations, a garbling scheme $\mathcal{G}$ is a tuple of four functions $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev})$. Let $f : \{0,1\}^n \rightarrow \{0,1\}^m$ denote the function to be evaluated securely. Here, Gb is a randomized *garbling function* that converts the function f (represented as a Boolean circuit) into a *garbled* circuit F. It also outputs encoding and decoding information e and d needed for encoding inputs and decoding the outputs. As such, $(F, e, d) \leftarrow\!\!\$\ \mathsf{Gb}(1^\kappa, f)$, where κ is the security parameter. The *encoding function* (En) encodes an input $x \in \{0,1\}^n$ using the encoding information e, which is the set of labels corresponding to the value of each bit in x; $X \leftarrow \mathsf{En}(e, x)$. The *evaluation function* (Ev) enables evaluation of the garbled circuit F over the garbled input X to generate the garbled output $Y \leftarrow \mathsf{Ev}(F, X)$, which is the set of labels corresponding to the output wires. Finally, the *decoding function* (De) decodes the output of the evaluation $y \leftarrow \mathsf{De}(d, Y)$.

Garbling involves generating two random labels L_1^w and L_0^w for each of its wires, representing the true and false value for the wire w. A number of optimizations have been proposed to reduce the size of a garbled circuit. One of them is the free XOR technique [43], which requires all wire labels to follow the form $L_1^w = L_0^w \oplus e_r$, where e_r is a string randomly chosen by Gb. This allows XOR gates in the circuit to be computed with only the input wire labels.

Typically, GCs are used for 2-party secure function computations where two parties with their respective private inputs x_1 and x_2 run the protocol such that, no party learns more than $f(x_1, x_2)$ for a public function f. The protocol works as follows. First, one of the parties, called the *generator*, uses the garbling function to generate $(F, e, d) \leftarrow\!\!\$\ \mathsf{Gb}(1^\kappa, f)$. Next, it encodes its input as $X_1 \leftarrow \mathsf{En}(x_1, e)$. The other party, called the *evaluator*, receives F and X_1 and also retrieves $X_2 \leftarrow \mathsf{En}(e, x_2)$ — encoding of its private input x_2 — using an oblivious transfer (OT) [62] protocol with the generator. Following this, the evaluator runs the garbled circuit to obtain $Y \leftarrow \mathsf{Ev}(F, (X_1, X_2))$. Finally, either party can decode Y to obtain the final output $y \leftarrow \mathsf{De}(d, Y)$.

A secure garbling scheme provides the following security properties [17]: (a) *Message obliviousness.* Given (F, X), an adversary learns nothing about x or y (beyond what is known from f). (b) *Input privacy.* Given (F, X, d), an adversary learns nothing about x beyond what is known from y and f. (c) *Execution authenticity.* Given a garbled input X, it is hard to find Y' such that $Y' \neq \mathsf{Ev}(F, X)$ and $\mathsf{De}(d, Y') \neq \perp$.

[1] As of Aug 2020, 417 out of 522 services on IFTTT are third-party that require a user to login and authorize access to IFTTT.

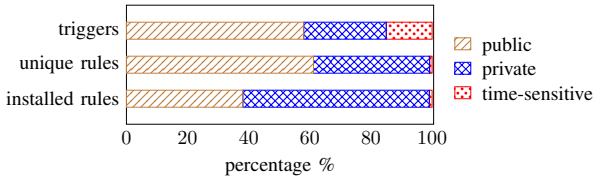

Fig. 2: Breakdown of triggers, rules, and installed rules in IFTTT based on their sensitivity levels.

We use these cryptographic primitives to design eTAP. In Section III, we analyze existing TAPs to understand what functions eTAP must support. We give the detailed protocol in Section V, with its security proven in Appendix A.

III. Analysis of Current Trigger-Action Systems

We analyze two popular commercial TAPs, IFTTT [8] and Zapier [11] with the following goals in mind: (1) understand the sensitive data that TAPs compute on; (2) establish that although TAPs offer a variety of operations on data, they are not arbitrary and will fit well in a garbled circuit framework; and (3) derive an abstract TAP computational model that will help ensure our system supports realistic functionality.

Types of sensitive information. The current trigger-action system design gives the cloud-based TAP complete access to trigger data. To better characterize the types of sensitive trigger data accessible to TAPs, we analyzed the IFTTT dataset mentioned in [50], by mapping each of its 320,000 IFTTT rules to one of the three trigger sensitivity levels defined by Bastys et al. [15] — public, private, and time-sensitive. Private triggers contain information like emails and calendar events, whereas public triggers contain information like news and weather reports. The time-sensitivity level means that private information exists in the availability of the trigger message. For example, considering the rule "IF I leave home, THEN turn off the WiFi," the TAP will learn whether the user leaves home depending on whether it receives a message from the trigger service. Fig. 2 shows a breakdown of sensitive trigger data according to how frequently they are used.

We observe that although a significant percentage (15%) of triggers and action APIs supported by IFTTT are time-sensitive, in reality, they are rarely used — only 0.8% of all available rules in IFTTT (or 0.9% of all installed rules) use a time-sensitive trigger. We also observe that, although there are fewer private triggers than public ones, private triggers are most frequently used — 61% of all installed rules contain a private trigger API. These APIs return private information like emails, messages, location traces, photos, sensitive files, medication lists, health information, etc. Thus, we design eTAP to protect the vast majority of private trigger information that people actually use in real-world rules. We do not currently provide confidentiality for time-sensitive information, but we outline possible approaches using standard techniques like cover traffic in Section VIII.

Operations on trigger data. IFTTT allows users to express computation on trigger data using *filter code* — small snippets of TypeScript with some restrictions (e.g., no I/O operations) [5]. Zapier rules contain two components: *filters* that compute a predicate on the trigger data, and *formatters* that modify the trigger data. Multiple filters and formatters can be chained together.

Type	Operation	Description
Bool	`x | a`	`x` OR `y`
	`x & a`	`x` AND `y`
	`! x`	NOT `x`
Num	`x < n`	Is `x` *less than* `n`?
	`x > n`	Is `x` *greater than* `n`?
	`x.mathop(n)`	Basic math ops. ($+, -, \times, \div$)
	`x.format()`	Format `x` into a string
Str	`x == s`	Does `x` *exactly match* the string `s`
	`x.contain(s)`	Does `x` *contain* the string `s`
	`x.startwith(s)`	Does `x` *start with* the string `s`
	`x.endwith(s)`	Does `x` *end with* the string `s`
	`x.split(d, i)`	Split `x` using delimiter string `d` and select the `i`-th substring
	`x.replace(s, t)`	Replace all occurrences of `s` in `x` with `t`
	`x.to_lowercase()`	Convert all characters in `x` to lowercase
	`x.truncate(n)`	Truncate `x` to size `n`
	`x.extract_phone()`	Extract the first phone number found in `x`
	`x.extract_email()`	Extract the first email address found in `x`
	`x.strip_html()`	Remove all HTML tags in `x`
	`x.html2markdown()`	Convert all HTML tags in `x` to Markdown
	`m.lookup(x)`	Look up the value for the key `x` in a user-provided map `m`
Any	`x == null`	Does `x` *exist*?
	`x.default(y)`	Set value of `x` to `y` if it does not exist

Fig. 3: Operations used in top 500 IFTTT rules with private triggers and all Zapier's function-dependent rules.

To understand the common operations in IFTTT, we again used the dataset of Mi et al. [50]. We selected the 500 most popular rules (based on user installation count) that are connected to private trigger APIs. Unfortunately, a challenge is that filter codes for IFTTT rules are not public. We therefore manually approximated the filter code for these rules by (1) estimating the functionality of each rule based on their title and description, (2) examining the corresponding trigger/action APIs, and (3) deducing the operations that are required to convert trigger fields to action fields.

We also crawled the Zapier website for one day in October 2019 and collected all the publicly available rules that require computations on trigger data [6], [7]. We collected a total of 378 rules and extracted the operations used in those rules.

The operations we found in IFTTT and Zapier are shown in Fig. 3. Current garbled circuit libraries support a majority of these operations natively. The main challenge is string operations, for which we contribute a novel technique to convert deterministic finite automata into Boolean circuits (Section V-E).

Execution model of trigger-action systems. Based on our survey of IFTTT and Zapier, we derive an abstract model of these trigger-compute-action rules. During rule setup on the client, the user typically specifies two functions — a *predicate* f_1, and a *transformation* f_2. These functions take the trigger data and some additional user-provided constants as input. The predicate function f_1 tests the trigger data for a condition to determine whether TAP should contact AS. The output of f_1

is either true or false. The transformation function f_2 modifies the trigger data before sending the result to AS. Both f_1 and f_2 run inside the cloud-based TAP.

Let $x \in \mathcal{X}$ be the part of the trigger data on which TAP performs some computation, and $y \in \mathcal{Y}$ be the action data TAP sends to AS, where $\mathcal{X}$ and $\mathcal{Y}$ are the domains of the trigger and action data, respectively. Both x and y can be data structures that contain multiple fields. We find that TAPs do not modify some fields of trigger data such as large media files, but only forward them to AS. We denote such trigger data as payload v. Let $c_1, c_2 \in \mathcal{C}$ be the two user-provided constants for the functions f_1 and f_2, where $\mathcal{C}$ is the domain of the constants. On receiving (x, v) from TS, TAP executes

"if $f_1(x, c_1) = \mathsf{true}$, then send $(f_2(x, c_2), v)$ to AS"

For simplicity, we assume the domains of f_1 and f_2 to be the same. So, $f_1 : \mathcal{X} \times \mathcal{C} \to \{\mathsf{true}, \mathsf{false}\}$, and $f_2 : \mathcal{X} \times \mathcal{C} \to \mathcal{Y}$.

TAPs operate in two modes: (1) *polling mode*, where TAP contacts TS at a predefined frequency; (2) *push mode*, where TS sends a message to TAP when an event occurs. While our protocol will work with both models, we assume the push model in this paper as it is more efficient in general.

Example rule. We show how our abstract model can instantiate our previous example rule: "IF I receive an email containing the word 'confidential', then send me an SMS." The SMS should contain the address of the sender and the email's subject. Assume that TAP provides an operation to search over strings, called `contain`. The user sets up a rule by choosing its email provider as the trigger service, that sends a copy of every new email to TAP. The action service is an SMS provider that sends SMS to a user-provided number. The user then specifies the `contain` function to check for the string $c_1 =$"confidential" on the email's subject line, x. The transformation function f_2 creates the required data structure to send the SMS, for example, setting the recipient address as the user-provided phone number c_2 and the message body as the concatenation of the sender's address and the subject.

IV. Design Considerations for Providing Data Confidentiality in Trigger-Action Systems

Our goal is to protect the confidentiality of private data involved in trigger-action rules even if they are run on a malicious cloud-based TAP. In this section, we discuss our threat model, define our security and functionality goals, and explore the design space.

A. Threat Model and Functionality Goals

Fernandes et al. [30] first noted the security and privacy issues of a compromised TAP and the related attacker motivations. We adopt the same attacker model — TAP is *malicious*. Specifically, the attacker: (1) can monitor communications between TAP and the trigger/action services; (2) can arbitrarily deviate from the communication protocol by manipulating, delaying, or dropping the messages; (3) can modify TAP's internal storage and code that includes manipulating and deleting garbled circuits; (4) knows API details of trigger and action services; and (5) knows the functions that are being evaluated on TAP. As we use cryptographic techniques for our security guarantees, we assume that the attacker is computationally bounded.

We assume that the end-point services (trigger and action services) like Samsung SmartThings, Google Calendar, etc. are *semi-honest* — they will follow the protocol as specified, but try to glean more information than what they are entitled to know. This is in line with the trust model used by current TAPs. Also, if they are compromised, then the attacker can achieve its goals of accessing and manipulating user data independently of the trigger-action system. We also assume that TAP is not colluding with TS or AS. As discussed in Section II, third-parties own a large majority of trigger and action services and thus collusion with TAP is unlikely (for example, there is no incentive for LG or Google to collude with IFTTT to reduce the security of their users). Enforcement of the non-collusion condition can also be done via legal affidavits [32], [65] or techniques that involve using a trusted mediator who monitors the communications between the parties [12], [13].

Finally, we assume that the user trusts their client device. We observe that the attacker is motivated to compromise TAP because it will simultaneously be able to attack all users of the platform. An attack on the client device is not scalable to all users easily, and therefore, is less attractive.

Security goals. Under this threat model, we want two security properties for a trigger-action system:

Privacy: Each party should not learn other parties' data in a trigger-action rule. Specifically, TAP should not learn the trigger data (x, v), user-provided constants (c_1, c_2), and results of the computation (beyond what they already know from the definitions of the functions); the trigger service (TS) should not learn the user-provided constants (c_1, c_2); the action service (AS) should not learn the trigger data x or user-provided constants (c_1, c_2) beyond what is revealed to it after rule execution. Additionally, AS should not learn the output of transformation function f_2 or payload v when the predicate function f_1 evaluates to false.

Integrity: The attacker should not be able to modify any computations on private trigger data without being detected by AS. That is to say, TAP should not be able to trick AS into acting on illegitimate action data, such as delayed, replayed, or tampered messages that are not the result of proper evaluation of the rule. AS only accepts valid messages $y = f_2(x, c_2)$, where x is sent by TS within the last τ seconds (a configurable parameter).

Security non-goals. Denial of service is outside our scope. A compromised TAP can indeed drop all messages it receives from TS and not transmit any message to AS. Metadata and side-channel attacks are also outside our scope. For example, even if messages are encrypted, the compromised TAP can observe the timing of messages that arrive from a trigger service or go to an action service. Coupled with semantic

knowledge about the services, this might enable the attacker to determine the sensitive data in the rule even if it is encrypted. As discussed in Section III, this involves time-sensitive rules which are less used frequently in practice. eTAP protects the vast majority of sensitive trigger data for which encryption achieves strong security properties. Section VIII outlines standard approaches to protect metadata that we leave as future work.

Functionality goals. We want to achieve the security goals while respecting the following functionality goals: *(1) RESTful API for end-point services*. The end services should be able to design their APIs independently of each other, as they do currently. These APIs should be RESTful, have minimal computational overhead beyond running the API itself, and do not need to store data or state specific to different trigger-action rules. *(2) Maintain trigger-compute-action paradigm.* The design should run existing user-created rules without any changes and should maintain the key architectural aspects of current trigger-action systems. Notably, the rules should execute without requiring the client device to be online.

B. Design Space Exploration

We explore a few potential solutions occupying different points in the design space and discuss why they do not meet our functionality or security requirements.

Computation at the edges. The trigger service can run a user-supplied function over its private data, encrypt the result, and forward that to TAP. However, the trigger service has to support an execution infrastructure similar to AWS Lambda, significantly increasing the complexity and overhead of such services and exposing them to additional security risk due to executing third-party code. Furthermore, sensitive data in user-supplied constants (c_1, c_2) will be exposed in plaintext to the trigger service. For example, consider rule R7 from Fig. 9, which converts Slack mentions to Asana tasks (a project management tool). It requires users to provide a lookup table of project names. These are sensitive information that should not be revealed to Slack. Computation can also be moved to the action service, but the same issues exist there as well.

Secure hardware. It is possible to use hardware-based trusted execution environments (TEEs) or hardware security modules (HSMs) for computing the trigger data on TAP, while preserving confidentiality [66], [78]. Yet besides requiring hardware changes to the TAP servers, current TEEs suffer from fundamental security design issues [23], [54], [69].

Homomorphic encryption of the trigger data. During rule setup, the client can specify a symmetric key between the trigger and action service. The trigger service encrypts its data using this key before sending it to TAP. This will provide trigger data confidentiality and allow the TAP to compute directly on the encrypted data. However, only specialized schemes like linear homomorphic encryption and "somewhat" homomorphic encryption are practical [55], thus limiting expressivity. For reference, TFHE [10], a state-of-the-art library for fully homomorphic encryption, takes 4.45 seconds to compute an addition circuit, which is 3 orders of magnitude slower than our system as evaluated in Section VI. Additionally, protection against a malicious TAP would require zero-knowledge proofs [33] of computation that would further reduce efficiency.

Off-the-shelf secure multi-party computation. Secure multi-party computation (SMC) protocols allow multiple distrusting parties to compute a function over their private inputs [74]. However, efficient off-the-shelf SMC protocols do not fit our threat model — TAP is malicious, or architectural requirements — needing TC, TS, AS, and TAP to participate in a multi-round protocol during rule execution. Therefore, we adopt a core primitive of SMCs — garbled circuits — and modify it to our setting.

Secret sharing based SMC. Secret sharing is an alternative to garbled circuits for doing SMC. However, secret sharing-based protocols require intensive multi-round communication (e.g., for evaluating multiplication gates). Additionally, in such protocols every party has to do an equal amount of work, which will require invasive architectural changes to TS and AS. This violates our functionality goal. Finally, the malicious versions of these protocols are not efficient.

V. Design of Encrypted Trigger-Action Platform

In this section, we discuss eTAP's core protocols and analyze how we specialize garbled circuits to trigger-action systems. A high-level overview of eTAP is shown in Fig. 4, and the pseudocode is given in Fig. 5. Like a typical trigger-action system in Fig. 1, eTAP has four components: trusted client's device (TC), trigger service (TS), action service (AS), and a trigger-action platform (TAP). We describe below how our design modifies these four components while maintaining the trigger-compute-action paradigm.

Decentralized trust model. In the current trigger-action system design, users place all trust within a centralized cloud-based TAP. This design leaves open a large-scale security and privacy risk — a single compromise of the TAP will simultaneously compromise all users. To avoid this issue, eTAP borrows a design element from DTAP [30] and designates the user's client device (smartphone) as the root of trust. Each user *only* trusts their own smartphone and uses it to program trigger-compute-action rules. As the eTAP protocols are open-source, we envision a community of developers building client apps, much like we have apps for open protocols like SFTP, Telnet, etc. Thus, the eTAP cloud component and the client app are built and controlled by different entities. Therefore, the client app can still be trusted, even when the TAP is compromised. eTAP bootstraps its guarantees on top of this model. In eTAP, the trusted client (TC) is beyond just an interface — it stores some state (as we describe below) that is key to its operation.

A. Rule Setup (occurs on trusted client)

Like in existing trigger action systems, the user can configure a trigger-compute-action rule on the trusted client app (TC)

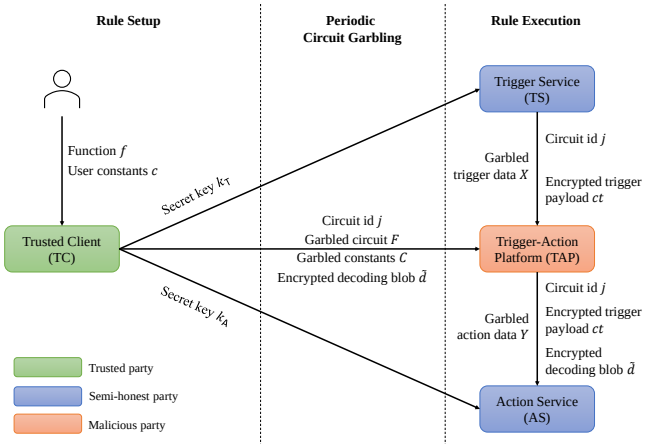

Fig. 4: Overview of eTAP.

using its click-through interface. The user selects a trigger in a trigger service (TS), a predicate f_1, a data transformation over the trigger data f_2, and an action in an action service (AS). The user also specifies any constants c if required.

TC sends the rule descriptions to TAP and helps the TAP negotiate OAuth tokens with TS/AS required for running the rule. In eTAP, unlike existing TAPs, TC shares with TS and AS two uniformly-generated secret keys $\boldsymbol{k}_T$ and $\boldsymbol{k}_A$, upon successful authorizations. The key $\boldsymbol{k}_T$ and $\boldsymbol{k}_A$ are tied to the specific trigger and action API for this user in TS and AS[2]. If a prior rule has already been set up with the same trigger or action API, then the corresponding OAuth authorization can be skipped and TC will reuse the previously generated $\boldsymbol{k}_T$ or $\boldsymbol{k}_A$. Once the rule is setup, TS and AS store the shared key materials; TAP stores the OAuth tokens; and TC stores the rule (f_1, f_2), the keys $(\boldsymbol{k}_T, \boldsymbol{k}_A)$, and the constants (c_1, c_2) provided by the user for the rule.

B. Circuit Garbling (periodic, occurs on trusted client)

Once the user creates a new rule, TC has to generate garbled circuit to enable secure evaluation of the functions on the (untrusted) TAP. TC generates garbled circuits corresponding to f_1 and f_2 and the associated encoding/decoding blobs. It uses the encoding blob to obtain the garbled labels for user-supplied constants. The decoding blob allows AS to decode the garbled outputs and to decrypt the payload. To ensure TAP does not learn or tamper with the decoding blob, TC encrypts it using $\boldsymbol{k}_A$. TC sends the garbled circuits, encoded constants, and encrypted decoding blob to TAP. TC identifies each instance of the garbled circuit using a monotonically increasing counter j. The circuit id j is initialized to zero if this is the first rule where the user uses the connected trigger API; otherwise, TC queries TAP for the circuit id that the connected trigger API is currently using. As garbled circuits cannot be reused, TC periodically repeats the above process.

[2]For better usability, current TAPs only acquire one OAuth token per service that can access all APIs in it [30]. eTAP can adapt to this model by exchanging a service-level key $\boldsymbol{k}_{\text{TS}}, \boldsymbol{k}_{\text{AS}}$, and derive the API-level keys $\boldsymbol{k}_T, \boldsymbol{k}_A$ from the hash value of $\boldsymbol{k}_{\text{TS}}, \boldsymbol{k}_{\text{AS}}$ and API URL, as required.

$\underline{\mathsf{CktGarbling}((f,c),(\boldsymbol{k}_T,\boldsymbol{k}_A,j))}$:
$e_s \leftarrow \mathsf{H}(\boldsymbol{k}_T \| j \| 0)$
$e_r \leftarrow \mathsf{H}(\boldsymbol{k}_T \| j \| 1) \vee 0^{\kappa-1}1$
$k_v \leftarrow \mathsf{H}(\boldsymbol{k}_T \| j \| 2)$
$e \leftarrow (e_s, e_r)$
$(F, L_0^{w_0}, \ldots, L_0^{w_m}) \leftarrow \mathsf{Gb}'(e, f)$
$d' \leftarrow (\mathsf{lsb}(L_0^{w_1}), \ldots, \mathsf{lsb}(L_0^{w_m}))$
$h \leftarrow \mathsf{H}(L_0^{w_1} \| \ \ldots \ \| L_0^{w_m})$
$\tilde{s} \leftarrow\!\!\$\ \mathsf{E}(L_1^{w_0} \oplus \boldsymbol{k}_A, (j, k_v, e_r, d', h))$
$\tilde{h} \leftarrow \mathsf{HMAC}_{\boldsymbol{k}_A}(j \| L_0^{w_0})$
$\tilde{d} \leftarrow (\tilde{s}, \tilde{h})$
$C \leftarrow \mathsf{En}(e, c)$
Set $j = j + 1$
Return $j, F, C, \tilde{d}$

$\underline{\mathsf{TSExec}((x,v),(\boldsymbol{k}_T,j))}$:
$e_s \leftarrow \mathsf{H}(\boldsymbol{k}_T \| j \| 0)$
$e_r \leftarrow \mathsf{H}(\boldsymbol{k}_T \| j \| 1) \vee 0^{\kappa-1}1$
$k_v \leftarrow \mathsf{H}(\boldsymbol{k}_T \| j \| 2)$
$X \leftarrow \mathsf{En}((e_s, e_r), x)$
$t \leftarrow \mathsf{CurrentTime}()$
$ct \leftarrow\!\!\$\ \mathsf{E}(k_v, (t, v))$
Set $j = j + 1$
Return j, X, ct

$\underline{\mathsf{TAPExec}\big((j,X,ct),(F,C,\tilde{d})\big)}$:
$Y \leftarrow \mathsf{Ev}(F, (X, C))$
Return $j, Y, ct, \tilde{d}$

$\underline{\mathsf{ASExec}\big((j,Y,ct,\tilde{d}),\boldsymbol{k}_A\big)}$:
Parse Y as $(L^{w_0}, \ldots, L^{w_m})$
$(\tilde{s}, \tilde{h}) \leftarrow \tilde{d}$
$z \leftarrow \mathsf{D}(L^{w_0} \oplus \boldsymbol{k}_A, \tilde{s})$
If $z = \perp$ then
  $\tilde{h}' \leftarrow \mathsf{HMAC}_{\boldsymbol{k}_A}(j \| L^{w_0})$
  If $\tilde{h}' \neq \tilde{h}$ then Return $\perp$
  Else Return false
$(j', k_v, e_r, d', h) \leftarrow z$
If $j \neq j'$ then Return $\perp$
$y \leftarrow \mathsf{De}(d', (L^{w_1}, \ldots, L^{w_m}))$
$g \leftarrow \perp$
For $i \leftarrow 1$ to m do
  If $y_i = 0$ then $g \leftarrow g \| L^{w_i}$
  Else $g \leftarrow g \| (L^{w_i} \oplus e_r)$
$h' \leftarrow \mathsf{H}(g)$
$(t, v) \leftarrow \mathsf{D}(k_v, ct)$
$t' \leftarrow \mathsf{CurrentTime}()$
If $t' > t + \tau$ or $h \neq h'$ then
  Return $\perp$
Return y, v

Fig. 5: Circuit generation and rule execution protocols for eTAP. $L_1^{w_0}$ denotes the true label for the first output wire w_0, $L_1^{w_0} = L_0^{w_0} \oplus e_r$; τ is a threshold parameter used to ensure the freshness of a trigger. CktGarbling is run by TC asynchronous to the actual rule execution. The remaining three functions are run by TS, TAP, and AS during rule execution.

Although TC needs to transmit the garbled circuits and related information prior to rule execution (Fig. 4), we design eTAP such that TC does not need to be online during execution. TC generates and transmits GCs in batches at times when the smartphone is not being used (e.g., when charging at night). Our evaluation (Section VI-B) demonstrates that transmitting sufficiently many garbled circuits for a day generally takes less bandwidth than backing up a 1-minute Full HD video to a cloud drive. This achieves our design principle of keeping the client device offline during rule execution.

Note that in our setting, the generator of the garbled circuit is the smartphone client — a trusted entity. This is a key insight and design element that is possible due to the nature of our setting. This allows eTAP to use efficient semi-honest implementations of garbled circuits and achieve security in the presence of a malicious TAP.

Cryptographic Details. Without loss of generality, we assume $f_1 : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}$ and $f_2 : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^m$. For notational simplicity, we denote $f : \{0,1\}^n \times \{0,1\}^{2n} \rightarrow \{0,1\}^{m+1}$, such that $f(x,c) = f_1(x,c_1) \| f_2(x,c_2), c = (c_1, c_2) \in \{0,1\}^{2n}$. Additionally, let $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^\kappa$ denote a cryptographic hash function. The pseudocode of the circuit garbling is given by the CktGarbling function in Fig. 5.

Encoding blob. The encoding blob contains the information required to encode the trigger data and encrypt the trigger payload. It can be derived from the key $\boldsymbol{k}_T$ and the garbled cir-

cuit id j. TC generates three bitstrings $(e_s, e_r, k_v) \in \{0,1\}^{3\kappa}$, using the hash of $\boldsymbol{k}_T \,\|\, j$. The false labels of the input wires (as described below) are generated using a H with e_s as the random seed, and e_r is used as a global offset for the standard free-XOR optimization [43]. The least significant bit of e_r is set to 1 to enable the standard point-and-permute optimization [16], [77]. Thus $e = (e_s, e_r)$ constitutes the encoding information used for the garbling scheme's encoding function (En). The key k_v is used to protect the payload data v.

Garbled circuit. To generate the garbled circuit F for function f, the labels for every input wire w are computed as $L_0^w = \mathsf{H}(e_s \| w)$ and $L_1^w = L_0^w \oplus e_r$ (assuming wire index w is a fixed-length bitstring). The rest of the computation (generating labels of the non-input wires and garbling gates) proceeds as per standard techniques with optimizations, such as row-reduction [56] or half-gate [77].

Encrypted decoding blob. The decoding blob consists of information necessary for AS to decode the labels of output wires (that correspond to the action data y) and to decrypt the payload. Let the output wires be $(w_0, w_1, \ldots, w_m)$, where w_0 corresponds to the output wire of f_1, and the remaining m wires correspond to those of f_2. Following standard practice [16], the decoding information d contains the least significant bits (lsb) of the false label of each output wire $(\mathsf{lsb}(L_0^{w_0}), \ldots, \mathsf{lsb}(L_0^{w_m}))$. In eTAP, decoding information is slightly modified. First, the first bit, $\mathsf{lsb}(L_0^{w_0})$, of d is dropped to create d'. Second, the hash of all the false labels of f_2's output wires $h \leftarrow \mathsf{H}(L_0^{w_1} \,\|\, \ldots \,\|\, L_0^{w_m})$ is computed. Third, a decoding blob is created using d', h, the payload key k_v, the XOR offset e_r, and the current circuit id j. Next, the whole blob is encrypted using a symmetric-key encryption scheme E with a key derived from both $\boldsymbol{k}_A$ (the secret key shared with AS) and $L_1^{w_0}$ (the true label of f_1's output wire w_0) to obtain $\tilde{s} \leftarrow\!\!\$\ \mathsf{E}(L_1^{w_0} \oplus \boldsymbol{k}_A, (j, k_v, e_r, d', h))$. Additionally, an HMAC [45] of the false label of predicate f_1 is computed using $\boldsymbol{k}_A$ as $\tilde{h} \leftarrow \mathsf{HMAC}_{\boldsymbol{k}_A}(j \| L_0^{w_0})$. We use $\tilde{d}$ to denote the tuple $(\tilde{s}, \tilde{h})$. We explain the rationale behind these changes in Section V-D.

Encoded user constants. Using the encoding information e, TC computes the labels for constants c as $C \leftarrow \mathsf{En}(e, c)$.

To accommodate the above customization, we derandomize the garbling function Gb to Gb' that takes an encoding information e as an input and returns the garbled circuit F, as well as the false labels of every output wire. TC sends $(j, F, C, \tilde{d})$ to TAP and increments the circuit id j by 1.

C. Rule Execution (occurs on TAP; does not involve TC)

When new trigger data is available for a trigger API, TS will garble the input data and encrypt any payload data, using the encoding blob it computes from $\boldsymbol{k}_T$ and circuit id j (which is initialized to 0 when the API is first called). It then transmits the ciphertexts to TAP, which will lookup any rules that are connected to the trigger API (and user) and run the associated garbled circuits. TAP finally transmits the output of the evaluation (garbled action data) and the encrypted decoding blob to the corresponding API in AS, which can decode to the plaintext result using $\boldsymbol{k}_A$ (Fig. 4).

TS and AS only perform simple encoding and decoding of data — fixed functionality independent of the trigger-action rule semantics, thus maintaining their RESTful nature. We believe that TS and AS are well-motivated to support these additional operations, in exchange for enhanced security. Indeed, current end-point services are concerned about the privacy of user data. For example, GMail recently removed their IFTTT triggers citing security and privacy concerns [61].

In our setting, the full evaluation of the garbled circuit is split between the untrusted TAP that executes the circuit to produce garbled output labels and the semi-honest AS that decodes the plaintext result from the labels. This, in combination with the trusted generator, allows eTAP to efficiently achieve the execution authenticity property of GCs using a hash function (Section V-D), even when TAP itself is malicious. We omit the standard OAuth steps that occur during execution, which the reader can refer to [9] for details.

Cryptographic Details. TS's operations in the rule execution phase is function TSExec in Fig. 5. TS recomputes the encoding information $e = (e_s, e_r)$ and the payload key k_v from $\boldsymbol{k}_T$ and j. It then encodes the trigger data x using the garbling scheme's encoding function, producing $X \leftarrow \mathsf{En}(e, x)$, and encrypts the payload v under a symmetric-key encryption scheme with the key k_v to compute $ct \leftarrow\!\!\$\ \mathsf{E}(k_v, (t, v))$ where t is the current timestamp. Finally, TS forwards the message (j, X, ct) to TAP and increments j by 1.

Upon receiving a trigger message (j, X, ct), TAP retrieves the corresponding garbled circuit F, garbled constants C, and the encrypted decoding blob $\tilde{d}$ using the trigger API and the circuit id j. Next, TAP evaluates F to obtain the garbled action data $Y \leftarrow \mathsf{Ev}(F, (X, C))$ and forwards the tuple $(j, Y, ct, \tilde{d})$ to AS. Function TAPExec in Fig. 5 depicts this process.

After receiving a message from TAP, AS decrypts $\tilde{d}$ to obtain the decoding information, which will succeed only when f_1 evaluates to true (i.e. $L^{w_0} = L_1^{w_0}$). If AS is able to decrypt the decoding blob, it uses (d', k_v) to obtain the final output $(f_2(x, c_2), v)$ in plaintext. AS would terminate if the message from TAP is malformed (i.e., hash of labels is inconsistent or decryption fails) or stale (i.e., trigger timestamp is old). The function ASExec in Fig. 5 depicts this process.

D. Rationale for Novel GC Protocol & Security Analysis

eTAP adopts a customized GC-based protocol tailored to the needs of trigger-action platforms. This protocol is novel in the following ways: (1) By leveraging the structure and threat model of trigger-action systems, we can use efficient semi-honest implementations of GCs to obtain security against a malicious evaluator; (2) eTAP supports fixed-length string operations including matching, extraction, and replacement — common operations in trigger-action programs — using Boolean circuits only; (3) eTAP contributes an efficient technique to ensure authenticity on the evaluator's output (i.e., TAP) that requires only two hashes instead of the existing

standard approach that requires hashes for true and false labels for every output wire.

Our setting has four parties: TC generates the garbled circuit via Gb' and then, both TC and TS use $\mathsf{En}(e, \cdot)$ to encode their respective inputs. On the other hand, TAP evaluates the garbled circuit using $\mathsf{Ev}(F, \cdot)$ while AS decodes the plaintext output using $\mathsf{De}(d', \cdot)$. Thus, TC and TS jointly act as the "generator", and TAP and AS jointly emulate the role of the "evaluator" of a two-party computation setting. The evaluators (TAP and AS) in our setting do not have any private input, therefore, eTAP does not require any oblivious transfers. Trust assumptions of the constituent parties of the generators and evaluators are asymmetric. Among the generators, TC is fully trusted and TS is semi-honest; among the evaluators, AS is semi-honest and TAP is fully malicious. Recall, TS and AS do not collude with TAP. (See Section IV-A for the motivations behind these trust assumptions.)

Next, we highlight the changes we introduce in two-party GC protocol and the rationale behind those changes. We formally prove all security properties of our protocol in Appendix A.

(1) TC generates the encoding information deterministically from the shared secret key $\boldsymbol{k}_T$ and the circuit id j, so that TS can also generate it without any communication with TC during rule execution. This achieves our design goal of ensuring that TC can be offline during rule execution. We note that this change does not violate the input privacy guarantees of the GC (see Thm. A.1, A.3, A.4, and A.5).

(2) Recall that the decoding blob (which contains information to decode garbled action data and to decrypt payload) is encrypted using the bit-wise XOR of $\boldsymbol{k}_A$ and $L_1^{w_0}$ as the key. Thus, TAP cannot learn the decoding blob (it does not have $\boldsymbol{k}_A$, Thm. A.1). Only AS can successfully decrypt $\tilde{s}$ if it gets the true label of the output wire of f_1, $L_1^{w_0}$, from TAP; which can happen only when the predicate $f_1(x)$ evaluates to true. This meets our privacy requirement that AS should not learn $f_2(x, c_2)$ or v when $f_1(x, c_1) = \mathsf{false}$. We formally prove this in Thm. A.4 and A.5.

(3) eTAP ensures that the malicious TAP (evaluator) cannot tamper with the results of evaluation. To achieve this we add the following information to the decoding blob: $h = \mathsf{H}(L_0^{w_1} \parallel \ldots \parallel L_0^{w_m})$, the XOR offset e_r, and $\mathsf{H}(L_0^{w_0})$. Standard techniques to achieve this property require the hashes of both true and false labels for every output wire [77]. However, in eTAP, AS does not have access to the circuit F and the garbled inputs (X, C). This makes it safe to disclose e_r to AS (Thm. A.4, A.5). Thus, AS can compute $L_0^{w_1}, \ldots, L_0^{w_m}$ from the output labels (see ASExec in Fig. 5) and check whether TAP has returned forged labels for the output wires corresponding to f_2. The HMAC $\tilde{h}$ is used to ensure the authenticity of the first output wire corresponding to f_1, when it evaluates to false. Because of this structure, eTAP achieves efficient authenticity verification with two hash values (Thm. A.2). This modification, combined with trusted generator, allows us to use efficient semi-honest implementations of GCs while achieving security against a malicious evaluator (TAP).

(4) We use a circuit id j to synchronize between different parties (TS, TAP, AS) so that they evaluate the correct circuit. Malicious TAP can observe the circuit id (in plaintext) and can tamper with it. eTAP ensures that the AS will always be able to catch a lying TAP, and will never act on an incorrect circuit id j. (See the proof in Appendix A.) Metadata leaked due to learning j is outside the scope of this paper (Security Non-goals in Section IV-A). We discuss a potential solution in Section VIII.

E. Supporting TAP-Specific Operations with Garbled Circuits

While in theory any arbitrary function can be converted into Boolean circuits, and therefore can be computed using GCs, in practice they can be expensive. Via an analysis of existing real-world rules (Section III), we found that they involve well-defined and relatively simple Boolean and arithmetic operations — these are well-studied and efficiently supported by existing GC libraries.

However, we also found that many rules use string operations, such as matching regular expressions and extracting or replacing substrings. The corresponding Boolean circuits of these operations, unless properly designed, will be inefficient to execute using GC [52]. eTAP computes these string operations by first translating regular expressions into deterministic finite automatons (DFA) and then applying a novel approach to convert DFA to Boolean circuits that can be efficiently evaluated using GC and can be easily extended for substring extraction and replacement. We next describe how eTAP utilizes this approach to perform regular expression matching. Details on substring extraction and replacement are given in Appendix B. (Please refer to [22] for details of how to convert a regular expression into a DFA.)

Input and output representations. First, to avoid leaking the length of the string, every string field in the trigger data (and the action data) is padded to a fixed length bitstring. AS is responsible for removing the padding as necessary. The string is encoded into a fixed-length bitstring $\vec{x} = (x_1, \ldots, x_n)$ where $x_i \in \{0, 1\}$ before feeding into the encoding function En. Let the operation of the string be defined using the DFA Γ, which is represented as a five-tuple, $\Gamma = (\mathcal{S}, \Sigma, \delta, s_0, \mathcal{S}_F)$, where $\mathcal{S}$ is the set of states, Σ is the set of alphabets, s_0 is the initial state, and $\mathcal{S}_F$ is the set of final states. The transition function δ takes a state and an alphabet and returns the next state; therefore, $\delta : \mathcal{S} \times \Sigma \to \mathcal{S}$. Since every string is a bitstring, we have $\Sigma = \{0, 1\}$. Let $q = |\mathcal{S}|$ be the total number of states. Without loss of generality, we assume $\mathcal{S} = \mathbb{Z}_q = \{1, \ldots, q\}$.

Let $\vec{\delta}$ be the aggregated transition function that takes the entire string $\vec{x}$ as input and outputs the final state of the DFA,

$$\vec{\delta}(\vec{x}) = \delta(\ldots\delta(\delta(s_0, x_1), x_2), \ldots, x_n).$$

If $\vec{\delta}(\vec{x}) \in \mathcal{S}_F$, then $\vec{x}$ is accepted by the DFA, which means that the string matches the regular expression.

Converting DFAs into circuits. The main goal is to convert the transition function $t = \delta(s, x)$ into a Boolean circuit that

uses as few AND and OR gates as possible, to take advantage of the standard free XOR optimization [43].

Since both the states s and t are integers between 1 and q, one can choose to represent each state using $\log_2 q$ bits and find the truth table for δ. However, the resulting circuit would be hard to construct and minimize automatically. Instead, we encode each state as a bit-vector of size q using one-hot encoding. We use S to denote the encoding of a state $s \in \mathcal{S}$, and S^i represents the i-the bit of S, where $S^i = 1$ if $i = s$ and 0 otherwise. We can observe that when $S^i = 1$ and $x = 0$, $T^j = 1$ if and only if $\delta(i, 0) = j$ holds; Similarly, when $S^i = 1$ and $x = 1$, $T^j = 1$ if and only if $\delta(i, 1) = j$. Therefore, the output of the DFA becomes

$$\vec{\Delta}(\vec{x}) = \Delta(\ldots\Delta(\Delta(S_0, x_1), x_2), \ldots, x_n),$$

where Δ is the transition function that operates on the one-hot encoded states.

To represent the transition function Δ as a Boolean circuit, we first define two sets for each state s, P_0^s and P_1^s, where $P_b^s = \{i \mid \delta(i, b) = j\}$ for $b \in \{0, 1\}$. It holds that $T^j = 1$ if and only if either $x = 1$ and $\exists i \in P_1^j, S^i = 1$, or $x = 0$ and $\exists i \in P_0^j, S^i = 1$. That is to say, for $1 \le j \le q$,

$$\begin{aligned} T^j &= (x \wedge \bigvee_{i \in P_1^j} S^i) \vee (\neg x \wedge \bigvee_{i \in P_0^j} S^i) \\ &= (x \wedge \bigvee_{i \in P_1^j} S^i) \oplus (\neg x \wedge \bigvee_{i \in P_0^j} S^i). \end{aligned}$$

Because only one of the S^i will be 1 at any time, therefore the inner OR gates can also be replaced with XOR:

$$T^j = (x \wedge \bigoplus_{i \in P_1^j} S^i) \oplus (\neg x \wedge \bigoplus_{i \in P_0^j} S^i).$$

Note the above expression can be further simplified using the Boolean algebra property $(x \wedge a) \oplus (\neg x \wedge b) = ((a \oplus b) \wedge x) \oplus a$. Therefore, each bit in T requires at most one AND gate to compute. To run Γ over a string of length n, we need to apply transition function (Δ) n times, and thus the resulting circuit contains at most nq AND gates. Finally, to check if the final state is accepted by Γ, simply computing $\bigoplus_{j \in \mathcal{S}_F} S_n^j$ is sufficient.

We can observe that the size of the entire garbled circuit is $O(nq\kappa)$, on par with the communication cost of the state-of-the-art non-GC based customized approach [52]. However, being purely circuit-based, our approach allows functional conjugation with other operations and retains the same security properties of standard GC. We describe more details on how to extend this approach to perform substring extraction and replacement in Appendix B.

Supported functions. By incorporating the above techniques, we can use garbled circuit to efficiently compute common arithmetic operations, string operations, and dictionary lookup, which cover all but three functions listed in Fig. 3. We sketch the implementation details for each supported function in Appendix C. Based on our analysis in Section III, this set of operations enables eTAP to support 93.4% of the function-dependent rules published on Zapier and *all* of the 500 most popular rules on IFTTT.

It is possible to convert the remaining three unsupported functions (`format`, `strip_html`, and `html2markdown`) to Boolean circuits, as well, but the resulting circuits will be very large (for example, we need to build a full-blown parser to find HTML tags) and inefficient to evaluate. These functions are only used for formatting and do not require any sensitive user input. Thus, it is safe to run them on AS or TS directly with minor modifications to their APIs.

VI. Evaluation of eTAP

We prototyped eTAP and showed that it is competitive in performance with TAPs that do not provide any data privacy. We implemented the garbled circuit protocols described in Section V using EMP toolkit [71], a C++ library for multi-party computation. We build on EMP toolkit's semi-honest 2PC protocol. We use state-of-the-art optimizations (including free XOR [43] and half gates [77]) for improving efficiency and bandwidth. The security parameter is $\kappa = 128$. For other cryptographic operations we use Cryptography.io [2]. We use SHAKE-128 (a member of SHA-3 family [57]) as a cryptographic hash function, and AES in CBC mode with HMAC using SHA-256 as a semantically secure, non-malleable, robust symmetric-key encryption scheme. To convert regular expressions into DFAs we use the library dk.brics.automaton [53]. For all experiments, we used n1-standard instances in Google Cloud Platform configured with 2 vCPUs, 7.5 GB memory, and 1 Gbps network connection.

A. Performance of Basic Operations

eTAP supports Boolean, (integer) arithmetic, and string operations (which is sufficient to run most of the rules in Zapier and IFTTT). To evaluate the performance of these basic operations, we picked a set of representative operations from Fig. 3. For Boolean, we chose the AND operation since our circuits only contain AND and XOR gates, and the XOR gate can be computed without any encryption costs [43]. For numeric data, we selected comparison and multiplication between two 32-bit integers. For string operations, we divided them into two categories: operations that need regular expressions (`contain`, `replace`, `split`, and `extract_phone`) and those that do not (`lookup` and ==). We set the input `x` as a 100-character (800 bit) string, except for `lookup`, where we set `x` to a 10-character string. In the function `m.lookup(x)`, we set `m` to be a key-value store with 10 entries, where each key and each value is 10-characters long. For `x.replace(s, "")` and `x.contain(s)`, we set the `s` to a 4-character string. For `x.split(d, 0)`, we set `d` to be a single character.

While measuring the costs for above basic garbled circuit operations, we do not consider the overhead of other components like payload encryption, as they are independent of the operation. Fig. 6 shows the time required for each operation.

The circuit generation (at TC) and circuit evaluation (at TAP) take roughly the same amount of time for each operation, which is expected because they require roughly similar

	Operation	Computation time (ms) Client	TS	TAP	AS	GC size (KB)	# DFA states
Bool	`x & y`	4.0	3.7	3.7	3.9	0.03	–
Num	`x > n`	4.0	3.9	3.8	3.8	0.96	–
	`x * n`	4.0	3.7	4.0	3.7	31	–
Str	`x == t`	4.0	3.7	4.0	3.8	25	–
	`m.lookup(x)`	4.2	3.6	4.1	3.8	31	–
	`x.split(d,0)`	5.7	3.7	5.3	4.1	78	16
	`x.contain(s)`	7.8	3.9	7.4	3.9	123	47
	`x.replace(s,"")`	10.7	3.8	10.5	4.6	278	40
	`x.extract_phone()`	24.7	3.6	25.5	4.1	2191	108

Fig. 6: Execution time of different basic operations at the client (TC), the trigger service (TS), the action service (AS), and the TAP. We record the size of the garbled circuit sent from TC to TAP and the number of states in the DFA if applicable.

operations. Most of the Boolean, arithmetic, and some string operations (such as string equality or lookup) execute in less than 4 ms on the TAP. Complex string operations are also fast (takes less than 25 ms) under some reasonably sized inputs. TS and AS can encode/decode inputs in less than 5 ms.

We record the size of the garbled circuit ($|F|$) for each operation in the second-to-last column of Fig. 6. The garbled circuit F needs to be periodically transferred from the client to TAP and the size of the circuit changes significantly for different operations. Although for Boolean AND the circuit is only 31 bytes, the circuit size for a complex regular expression extraction, which is one of the most expensive operations we found, is quite large (2.2 MB). The size of garbled circuit increases with the number of states in the DFA and the length of the input string. The string replacement circuits (`replace`) are larger, about 2.25x, than their equivalent matching circuits (`contain`), even though the required DFA is larger for the latter operation. The lookup circuit is small (31 KB).

B. Performance of Running Complete Rules

Next, we measure the performance of eTAP on real-world rules. We first picked ten rules from the combined IFTTT and Zapier dataset we collected in Section III. These rules handle sensitive data of different sizes and cover a wide variety of operations (as noted Fig. 3). We list the rules with simple descriptions in Fig. 9. The first eight rules (R1-R8) involve frequently used functions, while the last two rules represent two rare but extreme scenarios. R9 requires a rarely-used `extract_phone` function, which appears only three times in our dataset and requires a complex regular expression to be evaluated over a long text, thus making it the most expensive rule to compute in our dataset. R10 is connected to a trigger that might have a large payload (videos), so its performance is more dependent on network bandwidth and latency.

For comparison, we built a skeleton version of each service following the current TAP model, where only plaintext data is exchanged and computed, as a baseline. We refer to this as PlainTAP. We used Python library Flask for the cloud component of TAP, as well as two RESTful servers that mimic the APIs provided in current trigger and action services. Two US-west instances hosting TS and AS, and two US-central instances for hosting TAP and the (simulated) TC. The network latency between US-west and US-central is 39 ms.

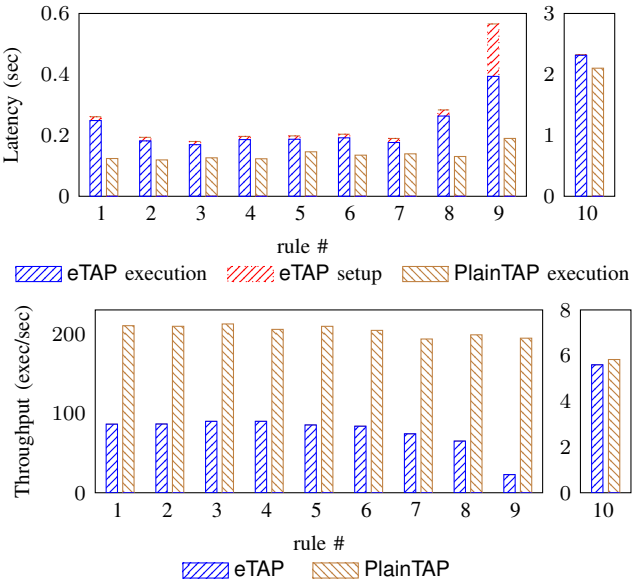

Fig. 7: Latency (top) and throughput (bottom) for running each of the rules (X-axis) in eTAP and PlainTAP.

Latency. The end-to-end execution latency measures the time between a trigger event (trigger data and payload are available to TS) and AS receiving plaintext output (Fig. 7). The latency, except R9 and R10, is below 260 ms. When compared to PlainTAP (Fig. 7, top), the execution latency for eTAP is 55% more on average. The majority of the latency overhead is due to the higher amount of data transfer in eTAP between TS and TAP (27-51 KB) and between TAP and AS (3-32 KB), which is nearly 128x more than what it would require in PlainTAP. We show the data transfer in the last two columns in Fig. 9. Given that TS, AS, and TAP are cloud-based services with high-bandwidth network links, the increase in data usage is reasonable. In addition, we list the time spent by TC to generate and upload a single circuit (as the red bar in Fig. 7, top). TC needs less than 12 ms to generate and transfer one circuit for most rules (except for R9, in which case it takes 172 ms). This metric represents the setup time for a new rule before it can be executed. In practice, TC can generate and upload circuits in bulk periodically at its convenience.

Throughput. We measured the throughput as the maximum number of executions per second by eTAP. We used Apache Bench [1] to compute the throughput, which simulates sending concurrent trigger messages to eTAP. We pre-computed the trigger labels to eliminate the bottleneck on TS. We gradually increased the concurrency level until the throughput saturated. We reported the maximum throughput of eTAP and PlainTAP in Fig. 7 (bottom). eTAP is capable of executing 65-90 rules of type R1-R8 per second on a single server. Compared to PlainTAP, for all but one (R9) rule, eTAP provides around 41% throughput of PlainTAP. In the worst case, when executing R9, eTAP's throughput reduces to 11% of PlainTAP.

C. Large-scale Evaluation

To better characterize the performance of eTAP under realistic workloads, we performed a large-scale evaluation where we randomly sampled 100 rules from our combined IFTTT and Zapier dataset. Out of the 100 sampled rules, 55 require computations on the trigger data. For rules with no computations, we simply treated the trigger data as payload and encrypted them inside ct.

Computation overhead on TC. In eTAP, the trusted client TC has to periodically generate and distribute the garbled circuits F, associated garbled constants C, and encrypted decoding blobs $\tilde{d}$ to TAP. For simplicity, we will use the term garbled circuit to denote the set $(F, C, \tilde{d})$. On average it takes 4.1 ms to generate one garbled circuit. Based on prior work [27], we assume that an average user has 26 rules installed and that each rule will be executed once every 15 minutes, which is the default interval used by IFTTT to contact its trigger services [50]. Therefore, we estimate that the TC of an average user needs to spend 10.2 seconds per day to generate $2,496$ circuits. Since the average circuit size is 25.3 KB, the estimated amount of data that TC has to send to TAP per day is 61.7 MB, which is less than the data required to back up 25 high-res photos or a 1-minute HD video (1920x1080 px @30 fps) to a cloud service [28], [35], a common task executed daily by modern smartphones.

Storage overhead. TAP needs to store all circuits uploaded by TC until they are executed. Based on the dataset in Section III, there are 12.4 million rules (counted by number of installations) running in IFTTT that are connected to private triggers. If we assume, conservatively, that all of them require computations, that each rule will be executed once every 15 minutes, and that each rule on average requires 25.3 KB of storage per execution based on the sampled rule set, the total storage overhead for using eTAP would be 28 TB. Given that cloud storage is inexpensive [14], the overhead is manageable. eTAP introduces little storage overhead to AS, TS, and TC. TS and AS only need to store a 16-byte key ($\boldsymbol{k}_T$ and $\boldsymbol{k}_A$) and the current circuit id j (4 bytes) for each user. TC needs to store the circuit id j and the keys for each service connected to the user's installed rules, since it can delete the circuits it generated after uploading them to TAP.

Latency and throughput. We first measured the end-to-end latency of running each rule individually and computed the average. The average latency of eTAP is 139 ms, which is similar to PlainTAP (110 ms). The increase in latency should be tolerable, considering the delays in current trigger-action systems are usually 1 to 2 minutes [50]. Then, we issued concurrent requests to trigger every rule at the same time and recorded the maximum throughput. The throughput of eTAP is 96 requests per second (RPS), which is 45% of the throughput of PlainTAP (211 RPS). Overall, we have shown that eTAP can run real rules with a modest performance impact.

VII. Related work

A few studies have investigated the security issues in IFTTT-like systems. Most closely related is the work of Fernandes et al. [30] where they first introduce the compromised TAP model, and then built DTAP, a system to prevent the misuse of stolen OAuth tokens. They focus only on the integrity problem. By contrast, our work subsumes DTAP by providing confidentiality to the private trigger data passing through TAPs and adding authenticity of trigger-compute-action rule execution.

Chiang et al. [24] recently propose Obfuscated TAP that handles metadata attacks. They propose techniques to hide trigger data arrival patterns and the types of trigger and action services from the untrusted TAP. Their work also performs end-to-end encryption of trigger data but cannot support computations. In contrast, eTAP focuses on protecting sensitive trigger data while allowing computation — a common use-case in real-world rules (e.g., filter codes in IFTTT).

Bastys et al. [15] classify the sensitivity of IFTTT's trigger and action services and show that 30% of IFTTT's apps may violate privacy by exfiltrating private information to a third-party. Xu et al. [73] analyze how much private data can be harvested by TAPs. They demonstrate that IFTTT has access to more data than necessary. For example, IFTTT monitors devices even if they do not trigger actions. This motivates our work in protecting all information from a malicious TAP.

A popular line of work investigates the semantics of rules and how they violate security policies or interfere with each other. Surbatovich et al. [68] present an empirical study of IFTTT apps and categorize the apps with respect to potential security and integrity violations. Wang et al. [70] design iRuler that uses SMT techniques to discover inter-rule vulnerabilities. This work is orthogonal to ours as it deals with rule semantics and the TAP is considered trusted. By contrast, our work protects trigger data from a malicious TAP.

Cryptographic techniques for secure computation. There is a large body of work on privacy-preserving outsourced computation. Garbled circuit is a particularly popular approach [21], [36], [44], [46], [67]. However, most practical approaches tend to be application-dependent [20], [42]. Since our setting differs from a generic multi-party setting (as discussed in Section V, we needed to develop a customized protocol).

For evaluation of string operations in a secure two-party computation setting, Mohassel et al. [52] introduced Obliv-DFA, a custom non-GC based protocol that only supports regular expression matching. Extending Obliv-DFA to substring extraction and replacements would not be possible without drastically changing the protocol and incurring significant overhead. eTAP, on the other hand, supports string operations through a novel and efficient purely circuit-based approach. This allows functional composition with other operations such as substring extraction/replacements and simple transfer of security properties of GC.

VIII. Discussion and Limitations

Security against metadata leakage. Some rules reveal sensitive information just because they are executed. For example, consider the rule: "IF I leave home, THEN turn off the WiFi." TS sends a message to AP only when the user leaves the home. In our threat model, TAP knows the rule semantics. Therefore, when TAP observes a message from this particular TS, it will learn that the user has left the home. Such metadata leakage from side-channels is hard to prevent cryptographically. Recent work [24], [73] has applied *cover traffic* to protect time-sensitive information in trigger-action rules by hiding the real trigger events among fake-but-identical ones. We discuss a simple modification to eTAP that uses cover traffic without requiring TC to generate new (fake) circuits.

TC generates a set of circuits and transmits them to TAP, as before. Let J denote circuit indices in this set. TS also internally keeps track of the set of circuit indices J' that have been used with *real* data. To send real data, TS picks random $j \in J \setminus J'$, updates $J' \leftarrow J' \cup \{j\}$, and continues as before (Section V). To send fake data, it picks random $j \in J$, and then sets the garbled trigger data to random bits. TAP executes the chosen circuit ID as before and sends output to AS. When fake data is evaluated on a circuit, the decryption at the AS will fail with very high probability and thus, it will ignore the message. We present an example to illustrate this approach. Assume TC generates two circuits with ids $J = \{j_1, j_2\}$ and TS has to send five events $e_1, e_2, \ldots, e_5$, among which only e_2 and e_3 are real. Following the scheme above, it transmits the following sequence of circuit ids to TAP: j_1, j_1, j_2, j_1, j_2. We see that j_1 is used multiple times: first for e_1, then for e_2, and finally for e_4. TAP will notice that j_1 circuit was executed thrice, but it cannot distinguish which of these executions was on real data.

This approach is secure due to two reasons: (1) TAP cannot distinguish between executions on real or fake data due to the garbling procedure we use in eTAP [77] (see Proposition 1 for a proof-sketch); and (2) TAP cannot learn anything from circuit usage statistics because of how TS selects j. In addition, circuits can be executed multiple times but *at most only one* of them will be on real data. Such re-evaluation of circuits on random data does not affect GC security properties [17].

Integrating with existing Trigger-Action Systems. We contribute a clean-slate redesign for trigger-action platforms providing data confidentiality from the ground up. As such, it is not immediately backward compatible. However, eTAP's design attempts to minimize these required changes as follows:

First, we create a new TC, a mobile app that users must install on their phones to interact with eTAP. The app mimics the user experience that trigger-action platforms like IFTTT or Zapier currently offer. For example, the user clicks on buttons in a wizard-style user interface to program a rule. TC transparently generates keys and GCs in the background and shares them with TS, TAP, and AS (accordingly) — the user does not have to take any additional action.

Second, the existing TS and AS need to adapt to eTAP protocol. Specifically, both need to communicate with TC to receive keys ($\boldsymbol{k}_T, \boldsymbol{k}_A$). Additionally, TS has to send encoded labels to TAP instead of plaintext trigger data, and AS has to run the decoding function on circuit output (Fig. 5). We have built a library that trigger/action services can use to upgrade their APIs to perform the above operations.

Third, TAP has to evaluate GCs. It also has to cache circuits it receives from TC. We observe that TAP is already setup to perform these tasks — executing code at large scale and managing user-specific data. Although this incurs a resource cost, we believe that it is acceptable given the strong confidentiality and integrity guarantees our work provides.

Rule semantics. A malicious TAP can learn about a user's automation patterns using its knowledge of rule semantics. Although we encrypt the trigger data, TAP can still observe the source endpoint of the trigger data and the destination of the encrypted result. As future work, we envision using results from anonymity networks like Tor [29] to hide the sources (trigger service) and destinations (action service) of messages.

Circuit id synchronization. eTAP requires TC and TS to synchronize on the circuit id j. TAP in eTAP cannot execute a rule if the j specified by TS is not present in its database of GCs sent by TC. This can happen, for example, if TC fails to generate circuits for a certain day due to technical glitches, but TS continues to generate trigger data. We do not want TS to support additional APIs to inform TC about its current circuit id j. Instead, we can rely on TAP to provide this information. TS attaches an encrypted (using $\boldsymbol{k}_T$) blob containing the circuit id and the timestamp to TAP along with other data during rule execution. TAP forwards that blob to TC on request from TC. Thus TC can learn the current value of j and can detect if TAP sends a stale message.

Loss of the trusted client (TC). TC in our setting is the "root" of trust for generating garbled circuits. TC can be an app running on user's personal mobile device. However, the app has to store a number of important states necessary for continued execution of a rule, such as $\boldsymbol{k}_T, \boldsymbol{k}_A$, OAuth tokens, j, f, c, etc. Therefore, the states on the trusted client must be preserved in case the device is lost. We can use standard cloud-based solutions to back up the states. For example, the states can be encrypted under a user's password and backed up in a cloud drive. The client can recover the states and continue to operate on a new device once the user connects their cloud drive accounts.

Circuit usage feedback. Different rules execute at varying rates. TAP can monitor rule execution frequency to make predictions about future circuit usage and optimize the number of circuit generations and transmissions. TAP can lie about these statistics; however, it does not affect on the security of eTAP. We leave its implementation to future work.

Acknowledgements

We thank the anonymous reviewers for comments that helped improve the paper. This work was supported in part by the University of Wisconsin-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. This work was also partially supported by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

References

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Cryptography.io. https://cryptography.io.

[3] 2017 Cybersecurity Incident & important consumer informationlist of data breaches. https://www.equifaxsecurity2017.com/consumer-notice/, 2017.

[4] Be more productive. Automatically. https://flow.microsoft.com/en-us/, 2020.

[5] Creating applets - ifttt platform. https://platform.ifttt.com/docs/applets#creating-applets, 2020.

[6] Filter by zapier integrations. https://zapier.com/apps/filter/integrations, 2020.

[7] Formatter by zapier integrations. https://zapier.com/apps/formatter/integrations, 2020.

[8] IFTTT: If This Then That. https://ifttt.com, 2020.

[9] Service api requirements - ifttt platform. https://platform.ifttt.com/docs/api_reference#service-authentication, 2020.

[10] Tfhe: Fast fully homomorphic encryption library over the torus. https://github.com/tfhe/tfhe, 2020.

[11] Zapier: Connect your apps and automate workflows. https://zapier.com/home, 2020.

[12] J. Alwen, J. Katz, Y. Lindell, G. Persiano, a. shelat, and I. Visconti. Collusion-free multiparty computation in the mediated model. In S. Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 524–540, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[13] J. Alwen, A. Shelat, and I. Visconti. Collusion-free protocols in the mediated model. In D. Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, pages 497–514, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] Amazon. Pricing for Elastic Block Storage. https://aws.amazon.com/ebs/pricing/, 2020.

[15] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[16] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. Association for Computing Machinery.

[17] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796, 2012.

[18] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology – ASIACRYPT 2000*, pages 531–545. Springer, 2000.

[19] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA, 1993. Association for Computing Machinery.

[20] J. Bringer, M. Favre, H. Chabanne, and A. Patey. Faster secure computation for biometric identification using filtering. In *2012 5th IAPR International Conference on Biometrics (ICB)*, pages 257–264. IEEE, 2012.

[21] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. *Journal of Computer Security*, 24(2):137–180, 2016.

[22] C.-H. Chang and R. Paige. From regular expressions to dfa's using compressed nfa's. In *Annual Symposium on Combinatorial Pattern Matching*, pages 90–110. Springer, 1992.

[23] 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 *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 142–157, 2019.

[24] Y.-H. Chiang, H.-C. Hsiao, C.-M. Yu, and T. H.-J. Kim. On the privacy risks of compromised trigger-action platforms. In *ESORICS*, 2020.

[25] C. Cimpanu. Millions of exim servers vulnerable to root-granting exploit. https://www.zdnet.com/article/millions-of-exim-servers-vulnerable-to-root-granting-exploit/, 2019.

[26] C. Cimpanu. Stack overflow hacker went undetected for a week. https://www.zdnet.com/article/stack-overflow-hacker-went-undetected-for-a-week/, 2019.

[27] C. Cobb, M. Surbatovich, A. Kawakami, M. Sharif, L. Bauer, A. Das, and L. Jia. How risky are real users'{IFTTT} applets? In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 505–529, 2020.

[28] S. Costello. How much video can you record on an iphone? https://www.lifewire.com/how-much-video-can-iphone-record-2000304, 2020. Accessed: 2020-06-02.

[29] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM '04, page 21, USA, 2004. USENIX Association.

[30] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized action integrity for trigger-action iot platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[31] T. K. Frederiksen, T. P. Jakobsen, and J. B. Nielsen. Faster maliciously secure two-party computation using the gpu. In *International Conference on Security and Cryptography for Networks*, pages 358–379. Springer, 2014.

[32] I. Giacomelli, S. Jha, M. Joye, C. D. Page, and K. Yoon. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In B. Preneel and F. Vercauteren, editors, *Applied Cryptography and Network Security*, pages 243–261, Cham, 2018. Springer International Publishing.

[33] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. Association for Computing Machinery.

[34] A. Hern. Uber employees 'spied on ex-partners, politicians and beyoncé', 2016. https://www.theguardian.com/technology/2016/dec/13/uber-employees-spying-ex-partners-politicians-beyonce.

[35] S. Hollister. The iphone 6s camera is a huge storage hog (but it might be worth it). https://www.cnet.com/news/iphone-6s-camera-filesizes-4k-live-photos-hdr/, 2015. Accessed: 2020-11-23.

[36] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*. USENIX Association, 2011.

[37] IFTTT. How people use IFTTT today. https://ifttt.com/blog/2016/11/connected-life-of-an-ifttt-user, 2016.

[38] IFTTT. Terms of Use. https://ifttt.com/terms, 2018.

[39] IFTTT. IFTTT: Creating Applets. https://platform.ifttt.com/docs/applets, 2020.

[40] IFTTT. IFTTT: Number of Users and Online Services. https://platform.ifttt.com/plans, 2020.

[41] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[42] V. Kolesnikov, A. Sadeghi, and T. Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 21(2):283–315, 2013.

[43] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 486–498. Springer, 2008.

[44] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[45] H. Krawczyk, M. Bellare, and R. Canetti. Rfc2104: Hmac: Keyed-hashing for message authentication, 1997.

[46] B. Kreuter, A. Shelat, and C. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*, pages 285–300. USENIX Association, 2012.
[47] D. Lee. Uber concealed huge data breach, 2017. http://www.bbc.com/news/technology-42075306.
[48] Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *International Conference on Security and Cryptography for Networks*, pages 2–20. Springer, 2008.
[49] I. Lunden. IFTTT raises $24M led by Salesforce to expand its platform to 'connect everything'. https://techcrunch.com/2018/04/26/ifttt-raises-24m-led-by-salesforce-to-expand-its-platform-to-connect-everything/, 2018.
[50] X. Mi, F. Qian, Y. Zhang, and X. Wang. An empirical characterization of ifttt: ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference*, pages 398–404, 2017.
[51] P. Mohassel. A closer look at anonymity and robustness in encryption schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 501–518. Springer, 2010.
[52] P. Mohassel, S. Niksefat, S. Sadeghian, and B. Sadeghiyan. An efficient protocol for oblivious dfa evaluation and applications. In *Cryptographers' Track at the RSA Conference*, pages 398–415. Springer, 2012.
[53] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2017. `http://www.brics.dk/automaton/`.
[54] 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.
[55] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. Association for Computing Machinery.
[56] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *EC*, 1999.
[57] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. 2015.
[58] Oauth 2.0. https://oauth.net/2/, 2018.
[59] D. Palmer. These hackers broke into 10 telecoms companies to steal customers' phone records. https://www.zdnet.com/article/these-hackers-broke-into-10-telecoms-companies-to-steal-customers-phone-records/, 2019.
[60] A. Peterson. eBay asks 145 million users to change passwords after data breach. https://www.washingtonpost.com/news/the-switch/wp/2014/05/21/ebay-asks-145-million-users-to-change-passwords-after-data-breach/, 2014.
[61] J. Porter. Elevating trust in our api ecosystem. https://developers.googleblog.com/2018/10/elevating-user-trust-in-our-api.html, 2018.
[62] M. O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. https://eprint.iacr.org/2005/187.
[63] Reuters. Database of 191 million U.S. voters exposed on Internet. https://www.reuters.com/article/us-usa-voters-breach-idUSKBN0UB1E020151229, 2015.
[64] P. Rindal and M. Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 297–314, 2016.
[65] A. Roy Chowdhury, C. Wang, X. He, A. Machanavajjhala, and S. Jha. Cryptε: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 603–619. ACM, 2020.
[66] S. Schoettler, A. Thompson, R. Gopalakrishna, and T. Gupta. Walnut: A low-trust trigger-action platform, 2020. https://arxiv.org/pdf/2009.12447.pdf.
[67] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE, 2015.
[68] 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.
[69] 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 *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.
[70] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1439–1453, New York, NY, USA, 2019. Association for Computing Machinery.
[71] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.
[72] Wired. Hack Brief: 4-Year-Old Dropbox Hack Exposed 68 Million People's Data. https://www.wired.com/2016/08/hack-brief-four-year-old-dropbox-hack-exposed-68-million-peoples-data/, 2016.
[73] 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*, 7:63457–63471, 2019.
[74] A. C. Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.
[75] A. C.-C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.
[76] F. Yao and Y. Yin. Design and analysis of password-based key derivation functions. In *Topics in Cryptology – CT-RSA 2005*, pages 245–261. Springer, 2005.
[77] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.
[78] 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. Security Analysis of eTAP

In this section, we show that eTAP meets the security goals outlined in Section IV-A by providing concrete security definitions and proofs. We assume the adversaries are probabilistic polynomial time (ppt) — they run in time polynomial in security parameter κ. The garbled circuit protocol $\mathcal{G}$ used in eTAP provides *output privacy*, *message obliviousness*, and *execution authenticity*. The encryption scheme $\mathcal{E}$ is IND-CCA secure. We model the hash function H as a random oracle [19]. Let $\mathsf{negl}(\cdot)$ to be a negligible function.

We prove the security of each component of eTAP, namely TAP, TS, and AS, separately. The security games are defined in Fig. 8.

Security against malicious TAP. Following our threat model, we assume the TAP is compromised and *malicious*. The security definitions we expect from eTAP are as follows.

Obliviousness. We define the obliviousness property of eTAP by the security game $\mathsf{Obliv}_{\mathcal{A}}^{\mathsf{etap}}$ as shown in Fig. 8. Informally, $\mathcal{A}$ despite arbitrarily deviating from the protocol should not know anything about the user-provided constants c, the trigger data x, v, and the output of the function $y \leftarrow f(x)$.

Theorem A.1 (TAP Obliviousness)**.** *For any ppt adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the $\mathsf{Obliv}_{\mathcal{A}}^{\mathsf{etap}}$ game is negligible.*

$$\Pr\left[\,\mathsf{Obliv}_{\mathcal{A}}^{\mathsf{etap}} = 1\,\right] \leq 1/2 + \mathsf{negl}(\kappa)\,,$$

Proof: The proof of this theorem follows directly from the *message obliviousness* security guarantee of garbled circuits $\mathcal{G}$ [77] and the semantic security of the encryption scheme $\mathcal{E}$. As such, the attacker learns nothing about (x, v, c) from (X, C, ct). First, note that the game $\mathsf{Obliv}_{\mathcal{A}}^{\mathsf{etap}}$ is equivalent to

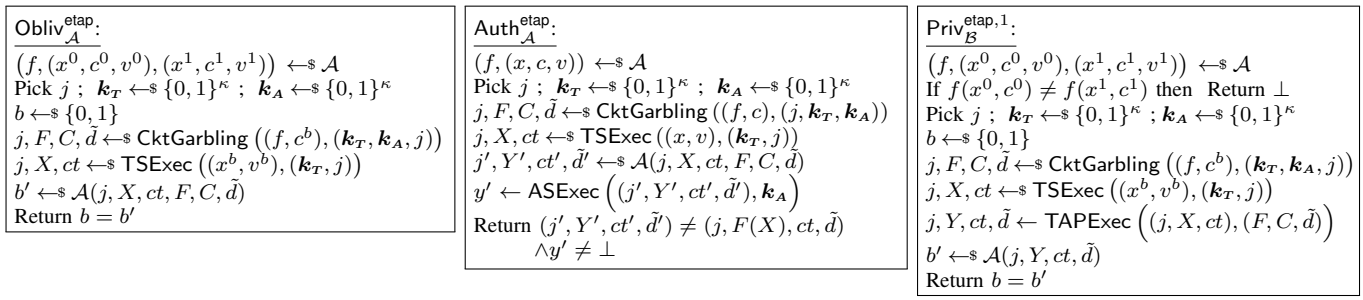

Fig. 8: Security games for eTAP.

the game $obv.sim_{\mathcal{S}}$ [17] in [77]. Now, consider the simulator $\mathcal{S}$ as presented in Fig. 3 in [77]. In our setting, $\mathcal{S}$ is used by TC and TS to generate $(\hat{F},\hat{X},\hat{C})$ which is then used for the rest of the computation. Hence the obliviousness of (x,c) follows directly from the corresponding proof (game $obv.sim_{\mathcal{S}}$) presented in [77] assuming the random oracle model for H [19]. The indistinguishability of ct^b follows trivially from the semantic security guarantee of the encryption scheme, thereby concluding our proof.

We achieve security against a malicious TAP even with a GC implementation for the semi-honest model. Recall that the "generators" — the trusted client (TC) and the trigger service (TS) — in eTAP are at least semi-honest. Hence, a valid garbled circuit for the correct function f is always generated (as TC is trusted), and all inputs are correctly encoded (since TS is semi-honest and the "evaluators" TAP and AS have no input). Thus, the only way a malicious TAP can compromise the security of eTAP is by forging an inauthentic output label or by replaying, delaying, or dropping a message. We discuss eTAP's resilience to such attacks next.

Authenticity. The security guarantee *authenticity* ensures that no ppt adversary can create a garbled output $Y' \neq Y$ such that AS acts on Y' (that is to say ASExec outputs anything but $\perp$ or false). The formal definition is given by the security game $\mathsf{Auth}^{\mathsf{etap}}_{\mathcal{A}}$ as shown in Fig. 8.

Theorem A.2 (TAP authenticity)**.** *For any ppt adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the game $\mathsf{Auth}^{\mathsf{etap}}_{\mathcal{A}}$ is negligible,*

$$\Pr\left[\,\mathsf{Auth}^{\mathsf{etap}}_{\mathcal{A}}=1\,\right] \leq \mathsf{negl}(\kappa)\,.$$

Proof: The proof follows from the non-malleability guarantee (IND-CCA) of the encryption scheme $\mathcal{E}$, execution authenticity of $\mathcal{G}$ [77], and the collision resistance of the hash function H. For the rest of the proof, consider the simulator $\mathcal{S}$ in [77] which additionally returns $h = \mathsf{H}(L_0^{w_1} \parallel \ldots \parallel L_0^{w_m}), e_r$ and $L_0^{w_0}$. TC uses this additional information to generate the decoding blob $\tilde{d}$. Similarly, the function in De is changed to that of ASExec.

Case I - Authenticity of $y_1 = f_1(x,c)$.
Note that $\tilde{s}$ is encrypted under a key derived from $\boldsymbol{k_A}$ and $L_0^{w_0}$. Hence, from the semantic security of the encryption scheme, TAP does not have access to e_r since it does not know $\boldsymbol{k_A}$ by design. Thus, in case $y_1 = \mathsf{false}$, TAP has access only to the false label $L_0^{w_0}$ and thereby cannot cheat AS. On the other hand, if $y_1 = \mathsf{true}$, TAP can return some garbage value L' such that $\mathsf{D}(L' \oplus \boldsymbol{k_A}, \tilde{d}) = \perp$. However, AS can detect this with the help of the HMAC. Moreover, TAP cannot send any of the hitherto unseen HMACs because it cannot obtain the output labels without access to the corresponding X (TS's trigger data).

Case II - Authenticity of $y_2 = f_2(x,c)$.
From the collision resistance of H, the only way TAP can cheat is by generating a label $L^{w_i}_{1-y_2[i-1]}$ for some wire $i \in [1,m]$ where $y_2[i]$ denotes the i-th bit of y_2. However, as discussed above, TAP cannot compute any other label other than the one obtained from $\mathsf{Ev}(F,X,C)$.

The rest of the proof follows an identical sequence of hybrids as the proof of Theorem 1 in [77] assuming the random oracle model for H.

Protection from altering the timing of rule execution. An adversary cannot forge a message that the AS will accept due to the strong authenticity guarantee of eTAP protocol. However, it can alter the execution time of a rule by deliberately dropping, delaying, or replaying messages. TAP can successfully drop a message without being detected by AS. However, this would fall under the denial-of-service attack which is beyond eTAP's scope (Section IV-A). eTAP also protects against replayed or delayed messages.

Every message from TS is timestamped as they are sent which AS can check before performing any action. Therefore, AS will reject a message — outputting $\perp$ — if the received message is delayed more than τ seconds (a parameter set by AS) since the time it was sent from TS. (See the function ASExec in Fig. 5.) We acknowledge that the TAP can replay any message for which $f_1(x,c) = \mathsf{false}$ without getting detected by the AS.

Nevertheless, this does not lead to any undesirable outcome in practice because in this case AS performs no action. Note that the above attack (replay of false labels) could have been prevented by keeping track of the last seen circuit id of each rule at AS. However, maintaining such state information would violate the RESTfulness of AS.

Tampering with circuit id j. The malicious TAP can modify the circuit id j — a unique identifier given to every instance of a garbled circuit for synchronization between TAP, TS, and AS — in whatever way they want to. But eTAP ensures AS will always be able to detect any such modification and rejects

the message from TAP (by outputting $\perp$). This is done by having TS include the circuit id j in the encrypted payload ct — that TAP cannot modify. AS verifies that value against the circuit id forwarded by TAP, and any mismatch results in execution termination. Though TAP cannot tamper with j without being detected, it could learn the popularity of certain rules by observing circuit id values (which are passed to TAP in plaintext to help find corresponding garbled circuit F to execute). We acknowledge that metadata attacks are a limitation in eTAP and we discuss a cover traffic approach to address them (Section VIII).

Security Analysis of TS and AS. We assume TS and AS are honest but curious. We define security as follows.

Theorem A.3 ($\mathsf{Priv}_{\mathrm{TS}}$)**.** *TS does not learn anything about the user constants* (c_1, c_2).

Proof Sketch. TS only receives from the client $\boldsymbol{k}_r$ and j, which it uses to compute the seed $e = (e_s, e_r)$. Thus, it can only learn the pairs of labels for all the input wires (including the ones for user constants) to the garbled circuit. TS, *by design*, does not have access to the client constants.

AS should not learn about the user constants and the trigger data beyond what is revealed from the output of the function f. Let $y_1 = f_1(x, c)$ and $y_2 = f_2(x, c)$. We also need to ensure that when the output of the predicate function $y_1 = \mathsf{false}$, AS does not learn the output of the function f_2 and the payload v. We formally state these properties, using the theorem below.

Theorem A.4 ($\mathsf{Priv}^0_{\mathrm{AS}}$)**.** *If* $y_1 = \mathsf{false}$*, then AS learns nothing about* (x, c, v) *other than what is revealed from* $y_1 = \mathsf{false}$.

Proof: To know the value of y_2, AS needs access to the decoding table d' (from the *obliviousness* guarantee of garbled circuits in [77]). AS will be able to do this only if it has access to $L_1^{w_0}$ (from the IND-CCA security of the encryption scheme). Note, $L_1^{w_0}$ is available to TAP, and subsequently to AS, only if $f_1(x) = \mathsf{true}$ [77]. In case TAP returns some garbage value other than $L_1^{w_0}$, the decryption still fails. Additionally, v is protected by the IND-CCA security of the encryption scheme.

Theorem A.5 ($\mathsf{Priv}^1_{\mathrm{AS}}$)**.** *If* $y_1 = \mathsf{true}$*, then for any ppt adversary* $\mathcal{B}$*, the probability that* $\mathcal{B}$ *wins the game* $\mathsf{Priv}^{etap,1}_{\mathcal{B}}$ *is only negligibly more than random guessing. That is,*

$$\Pr\left[\mathsf{Priv}^{etap,1}_{\mathcal{B}} = 1\right] \leq 1/2 + \mathsf{negl}(\kappa)\,.$$

Proof: The indistinguishability of ct^b follows from the semantic security of the encryption scheme. Now note that $\mathsf{Priv}^{etap,1}_{\mathcal{B}}$ is equivalent to $prv.sim_{\mathcal{S}})$ [17] in [77]. The rest of the proof is based on the proof for the corresponding game ($prv.sim_{\mathcal{S}}$) in [77]). In fact in our setting, the view of the $\mathcal{A}$ is a strict subset of that of the adversary presented in [77]. Specifically, our adversary $\mathcal{A}$ does not have access to the garbled inputs X^b, C^b and the garbled circuit F. Note that in the above game, a malicious TAP instead of outputting $(Y, ct) \leftarrow \mathsf{TAPExec}\,((X, ct), (F, C))$, could generate some arbitrary message. However, from the obliviousness property of garbled circuits (Thm. A.1, we know that this message has to be completely oblivious of (F, X, c) and hence the privacy guarantee is upheld trivially.

Proposition 1 (TAP Input Indistinguishability)**.** *For any ppt adversary* $\mathcal{A}$ *with access to a circuit garbled with the scheme in [77], the probability that* $\mathcal{A}$ *distinguishes between a valid garbled input and randomly generated input is negligibly more than random guessing.*

Proof Sketch. Following Fig. 2 in [77], it is clear that $\mathcal{A}$ cannot validate inputs to XOR gates. For AND gates, the fact that at most one valid label for each input wire is revealed to $\mathcal{A}$ and the correlated robustness of the hash function ensures that $F = (T_G, T_E)$ does not reveal information about the valid inputs.

B. Extracting and Replacing Substrings with Garbled Circuits

We now discuss how eTAP extends the regular expression matching technique described in Section V-E to extract and replace substrings.

Finding locations of matching substring. Given a regular expression pattern `p`, the goal is to find the starting and ending positions of the matching substrings.

Finding the ending positions can be achieved by applying the KMP algorithm [41] on the pattern p to convert it into a DFA (denoted by Γ), so that Γ will output an accepting state at the end of each matching substring. For example, if the pattern is `ab`, we will rewrite it as `.*ab` and convert the new pattern into DFA. Then we use our matching protocol (Section V-E) to run Γ on the input string $\vec{x}$. However, instead of only checking whether the final state S_n is an accepting state, we check every state $S_1, \ldots, S_n$ produced by Γ. We denote the resulting n-bit sequence as $e_1, \ldots, e_n$. If $e_i = 1$, it indicates that the i-th bit is the end of a matching substring.

Since a DFA can only report the end positions of matches end, we need another DFA to find the starting positions. We therefore compute a DFA Γ' on the reversed pattern p. If we run Γ' on the reversed input string, we get the beginning of the matching substring. Then, like the previous step, we run Γ' backward on $\vec{x}$ (by feeding from x_n to x_1) and check the type of every state to generate $b_n, \ldots, b_1$. If $b_i = 1$, it indicates that the i-th bit is the beginning of a matching substring.

Finally, we can find the locations of all matching substrings. That is, we need to compute another n-bit sequence $m_1, \ldots, m_n$ where $m_i = 1$ if and only if the i-th bit is part of a matching substring.

We can observe that $m_1 = b_1$ and for any i such that $2 \leq i \leq n$, m_i can be calculated as $m_i = b_i \vee (\neg e_{i-1} \wedge m_{i-1})$.

Extracting matching substring. To extract the matching substrings, we want to replace the characters in non-matching parts with the padding character (`0x00`). Therefore, the output string $\vec{y} = \{y_1, \ldots, y_n\}$ is computed by $y_i = m_i \wedge x_i$.

Replacing matching substring. In our dataset, all `replace(s,t)` functions are used with `t` set to empty string, so it is equivalent to removing the matching substring,

#	Rule description	Functions performed	GC size (KB)	Data transfer (KB) TS→TAP	TAP→AS
R1	Share your Tweets (excluding replies) in Slack	`! x[Text].startwith("@")`	0.2	43	20
R2	Get Slack notifications for new Twitter followers with more than 5,000 followers	`x[FollowerCount] > 5000`	1.0	29	3
R3	Copy New Events from Google Calendar into iOS Calendar	`x[StartTime] - x[EndTime]`	1.0	33	32
R4	Blink your lights when you receive email from a specific address	`x[Sender] == c`	5.8	29	3
R5	Send SMS messages for new Shopify orders	`x[Phone] != null; x[Phone].replace(" ", "")`	9.0	27	3
R6	Add new inbound emails as contacts in Ontraport	`x[SenderName].split(" ", 0); x[SenderName].split(" ", 1)`	30.5	34	13
R7	Create Asana tasks when new Slack messages start with `$request`	`x[Text].startwith("$request"); x[Text].replace("$request"); c2.lookup(x[Channel])`	92.4	29	4
R8	Save new liked Tweets with links to Pocket	`x[Text].contain("http")`	173.4	43	20
R9	Send SMS reminders for upcoming Google Calendar events	`x[Description].extract_phone()`	4,668.9	51	28
R10	Upload new videos in Google Drive to YouTube	`x[Filename].endwith("mp4|avi|mov")`	12.1	32,133	32,108

Fig. 9: Selected real-world rules for our experiments from both IFTTT and Zapier. We note the size of the corresponding garbled circuits, and the amount of data transferred from TS to TAP and TAP to AS during rule execution.

and thus the output string $\vec{y} = \{y_1, \ldots, y_n\}$ is computed by $y_i = \neg m_i \wedge x_i$.

However, for completeness, we will describe a protocol for the general case scenario where $|\mathtt{t}| > 0$, where `t` denotes the size of the string `t`. The output string size will be $n \times \frac{|\mathtt{s}|}{|\mathtt{t}|}$ since the TAP should not know which substring is matched and replaced and should assume all substrings can be replaced. When $|\mathtt{s}| \gg |\mathtt{t}|$ the sizes of the resulting garbled circuits will be unbearably large. Therefore, we purpose an alternative design approach where the actual replacement is processed in the action service: we replace the first character of each matching substring with some placeholder character, say `0xff`, and the rest with the padding character `0x00`, so the action service can invoke the following functions to complete the replacement: `y.replace("0x00", "");` `y.replace("0xff", t);` where `y` is the decoded output string. Note the first `replace()` is required regardless of our protocol, since it is needed for removing the padding from the input string.

We argue this approach does not break our security goal, revealing no additional trigger data that is not supposed to be revealed to the action service. If the replacement string t is considered sensitive the client can encrypt the replacement mapping with the 1 label of the output bit corresponding to $\bigvee_{i=1}^{n} m^i$, similar to how we protect d and k in Fig. 9.

Assuming an ASCII encoding and `0xff` as the placeholder character, we can compute the output string $\vec{y}$ using $y_i = s_{i-(i-1 \mod 8)} \vee (\neg m_i \wedge x_i)$, where the $i-(i-1 \mod 8)$-th bit is the first bit of the character that i-th bit belongs.

`x == s` and `x.startwith(s)`. A bit-wise comparison between `x` and `s` is performed up to the `min(len(x),`

C. Implementing Supported Function

In this appendix section, we describe how to implement each operation that appears in Fig. 3, except for Boolean and arithmetic operations, since existing GC frameworks like EMP toolkit [71] already provide built-in functions to efficiently translate them.

`len(s))` bit, and results are feed into a large AND gate as output. For `x == s`, We additionally check if the next remaining character in `x` or `s` is a padding character.

`x.endwith(s)` and `x.contain(s)`. These functions need to be first converted to a correspondingly regular expression and then matched against `x`.

`x.replace(s, t)`. We can apply the DFA replacement technique described in Appendix B directly for this type of functions.

`x.extract_phone()` and `extract_email()`. We apply the DFA extraction described in Appendix B by constructing appropriate regular expressions. However, as we need the matching results to be non-overlapping, one modification is needed : we can append `[^a-Z0-9]` to the regular expression and shift the final matching position forward by 1 character.

`x.split(d,i)`. Without loss of generality, we assume `d` is a single character. First we need to create two regular expressions, Γ_1 and Γ_2, to that output accepting states when the `i`-th and `i+1`-th occurrences of `d` is encountered. Once we have the starting and ending location of the substring, we can proceed with substring extractions.

`x.truncate(n)`. We can keep a variable counter `c` that gets increased after each bit in `x` is processed. And each output bit `y[i]` is computed by `x[i] & (n > c)`.

`x.tolowercase()`. Assume ASCII encoding, for each character in `x`, we first check if the last five bits are in the valid ranges; if so, we flip the sixth bit.

`m.lookup(x)`. First, we compare `x` with each key of `m`, and store the matching results into a `len(m)`-bit sequence. We denote this sequence as `b`. Then, the output `y` is computed iteratively by `y = (b[i] & v[i]) | (!b[i] & y)` as `i` ranges from 1 to `len(m)`.

D. Rules used for performance evaluation

The descriptions for the selected trigger-action rules we evaluated in Section VI-B are shown in Fig. 9.