Decentralized Action Integrity for Trigger-Action IoT Platforms

Earlence Fernandes
University of Washington
earlence@cs.washington.edu

Amir Rahmati
Samsung Research America
Stony Brook University
amir@rahmati.com

Jaeyeon Jung
Samsung
jae.jung@samsung.com

Atul Prakash
University of Michigan
aprakash@umich.edu

***Abstract*—Trigger-Action platforms are web-based systems that enable users to create automation rules by stitching together online services representing digital and physical resources using OAuth tokens. Unfortunately, these platforms introduce a long-range large-scale security risk: If they are compromised, an attacker can misuse the OAuth tokens belonging to a large number of users to arbitrarily manipulate their devices and data. We introduce *Decentralized Action Integrity*, a security principle that prevents an untrusted trigger-action platform from misusing compromised OAuth tokens in ways that are inconsistent with any given user's set of trigger-action rules. We present the design and evaluation of Decentralized Trigger-Action Platform (DTAP), a trigger-action platform that implements this principle by overcoming practical challenges. DTAP splits currently monolithic platform designs into an untrusted cloud service, and a set of user clients (each user only trusts their client). Our design introduces the concept of Transfer Tokens (XTokens) to practically use fine-grained rule-specific tokens *without* increasing the number of OAuth permission prompts compared to current platforms. Our evaluation indicates that DTAP poses negligible overhead: it adds less than $15ms$ of latency to rule execution time, and reduces throughput by 2.5%.**

I. Introduction

Trigger-Action platforms are a class of web-based systems that stitch together several online services to provide users the ability to set up automation rules. These platforms allow users to setup rules like, "If I post a picture to Instagram, save the picture to my Dropbox account." The ease of use and functionality of such platforms have made them increasingly popular [46], and several of them (*e.g.,* If-This-Then-That (IFTTT) [30], Zapier [18], and Microsoft Flow [5]) are on the rise. Furthermore, with the rise in popularity of connected physical devices like smart locks and ovens, we observe that many trigger-action platforms have started adding automation support for physical devices, making it possible for users to set up rules like: "If there is a smoke alarm, then turn off my oven" [22]. These platforms have privileged access to a user's online services and physical devices; thus they are an attractive target for attackers. If they are compromised, attackers can *arbitrarily* manipulate data and devices belonging to a large number of users to cause damage.

To better characterize this risk, we perform a brief survey of seven trigger-action platforms including an in-depth case study of IFTTT, a widely used platform with over 11 million users [29]. We find that trigger-action platforms support a wide variety of business and IoT use-cases using a logically monolithic design. This implies that if attackers compromise the platform, they will be able to leak OAuth tokens for all users. Indeed, compromise of web systems are commonplace. Prominent examples include Equifax [2], Target [16], US voters database [1], and Dropbox [9]. OAuth-specific attacks are on the rise as well. Yang *et al.* [51] showed that 41% of top 600 Android mobile applications, which use OAuth, are susceptible to remote hijacking, and the recent Google Docs OAuth-based phishing attack compromised one million users [36]. We observe that cloud services, even well-designed and tested ones, are not immune to persistent and sophisticated threats.

Furthermore, through API testing techniques, we find that in the case of IFTTT, the OAuth tokens it obtains for online services are overprivileged. For example, we find that it is possible to flash the firmware of a Particle chip, delete Google drive files, and turn off video surveillance in a MyFox smart home using IFTTT OAuth tokens. §III-B provides a more comprehensive analysis of these issues. We note that this risk of overprivilege is not isolated to IFTTT, but affects trigger-action platforms in general that use OAuth—incorrect OAuth scoping can lead to overprivilege—either trigger-action platforms may request broad scopes or the online services may only offer coarse-grained scopes. We conclude that beyond attackers misusing OAuth tokens of a compromised trigger-action platform, the overprivilege in the OAuth tokens extends the abilities of the attacker to invoke API calls that are outside the abilities of the trigger-action platform itself.

We show that it is possible to avoid this risk without losing the benefits of a cloud-based trigger-action platform. To that end, we introduce *Decentralized Action Integrity*. This security principle ensures that an attacker who controls a compromised trigger-action platform: (1) can only invoke actions and triggers needed for the rules that users have created; (2) can invoke actions only if it can prove to an action service that the corresponding trigger occurred in the past within a reasonable amount of time; and (3) cannot tamper with any trigger data passing through it undetected. To enable these security benefits, Decentralized Action Integrity makes use of four elements:

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23119
www.ndss-symposium.org

(1) Rule-specific tokens permit the bearer to execute a specific API call of an online service; (2) Timely and verifiable triggers ensure that the bearer of an OAuth token can invoke the action portion of a user-created rule only when it can prove that the triggering portion of the rule occurred within a reasonable amount of time in the past; (3) Data integrity ensures that an attacker cannot modify trigger data as it passes through the platform; and (4) Tokens are decentralized. A compromise of the platform does not leak tokens of all the users.

Decentralized Action Integrity is inspired by the end-to-end argument for system design by Saltzer, Reed, and Clark [39]. Rather than depending on the cloud service of a trigger-action platform, which can be compromised, to provide a proof that tokens were not misused, the principle places verification checks for misuse of OAuth tokens at the endpoints (*i.e.,* online services) of the system. Additionally, our work draws on the notions of Decentralized Trust Management [20], and the Kerberos Ticket-Granting Ticket system (§VIII illustrates these relationships in more detail).

We design, implement, and evaluate Decentralized Trigger-Action Platform (DTAP), the first trigger-action platform supporting Decentralized Action Integrity. Our design breaks down the currently monolithic structure of trigger-action platforms into an untrusted cloud service that executes user rules at large scale and a set of trusted client applications, where each user trusts their own client. While designing DTAP, there are a few challenges. First, rule-specific tokens can lead to a drastically increased number of OAuth permission prompts as users would have to login and approve an OAuth scope request every time they create a rule. The challenge is to gain the security of rule-specific tokens but maintain the current trigger-action platform experience where users approve OAuth requests only once during a setup phase for each online service. DTAP overcomes this challenge by using *Transfer Tokens* (XTokens). A small trusted client installed on the user's device uses an XToken to automatically obtain a rule-specific token, which it transmits to the cloud service for rule execution. Our implementation encrypts XTokens at rest using a hardware-backed keystore when available.

Second, DTAP requires the untrusted cloud service to prove to the invoked action service that a trigger has occurred within a reasonable amount of time in the past. As the cloud service can be compromised, a possible design is to have the trigger service communicate directly with the action service to verify the occurrence of a triggering event. However, this introduces an undesirable dependency between the action and trigger services. DTAP avoids that by using a lightweight cryptographic signature-based extension to the OAuth 2.0 protocol.

Our Contributions:

- We introduce Decentralized Action Integrity, a security principle that prevents an attacker from using stolen OAuth tokens in ways that are inconsistent with any given user's rules. We develop this principle based on a brief survey of seven trigger-action platforms, and an in-depth case study of IFTTT. Our analysis indicates that the logically monolithic designs of current trigger-action platforms coupled with overprivileged OAuth tokens pose a long-range large-scale risk to the digital and physical resources of users (§III, §IV).
- We designed and implemented Decentralized Trigger-Action Platform (DTAP), the first decoupled trigger-action platform supporting Decentralized Action Integrity, where users do not have to trust the cloud platform with highly-privileged access to their online services (§V). DTAP splits the logically monolithic trigger-action platform design into an untrusted cloud service that executes rules at scale, and a set of clients that help users create rules in a secure manner. DTAP is based on cryptographic extensions to the OAuth protocol that only allow the cloud service to execute user rules, even if it is attacker-controlled.
- We evaluate DTAP using various micro- and macro-benchmarks. Our evaluation shows that performance overhead is modest (§VI): Each rule requires less than $3.5KB$ additional storage space and imposes less than $7.5KB$ of transmission overhead per execution. DTAP adds less than $15ms$ of latency to rule execution time. For rules in trigger-action platforms, which typically send emails, SMSs, or invoke actions on physical devices or on online services over a network, we consider this additional latency to be acceptable. DTAP reduces throughput by 2.5% for rule execution.

We have designed Decentralized Trigger-Action Platform as an extension to the OAuth 2.0 protocol which is used by all current trigger-action platforms. Additionally, the protocol extensions do not require changes to the existing infrastructure of a trigger-action platform that is responsible for executing rules at large scale. These two aspects of the design indicate its wide applicability. Furthermore, our implementation provides a library that enables developers of online services to add a single line of code to gain the benefits of Decentralized Action Integrity. Although this represents a change to existing online services, we believe that DTAP is a valuable first step toward a clean-slate design of trigger-action platforms with strong security properties from the ground up.

II. Background: Trigger-Action Programs and Platforms

Trigger-Action platforms support stitching together various online services APIs such that end-users may write simple conditional programs. These simple programs often take the form "IF triggering condition, THEN take a specific action." Examples include "IF smoke alarm has fired THEN turn off the oven," and "IF NASA posts a new Instagram picture, THEN post it to my Dropbox." Table I shows a set of trigger-action platforms that we surveyed.[1] Based on our survey, we adopt a general terminology that describes the four main architectural components of such platforms:

- **Channel:** A channel represents part of an online service's set of APIs on the trigger-action platform. Users connect channels to their trigger-action platform accounts—a process that involves user authorization. For example, a user with a Facebook account must authorize the corresponding Facebook channel to communicate with her Facebook account. Channels communicate with online services using REST (Representational State Transfer) APIs operating over

[1]Our survey process was simple: create an account and create a single Rule, and then browse through the list of available online services that integrate with the platform.

Name	Purpose	# of Channels
IFTTT [30]	IoT/Business/Smart Home	500+
Zapier [18]	IoT/Business	750+
Microsoft Flow [5]	Business	156
Stringify [15]	IoT/Smart Home	74
Apiant [4]	IoT/Business	15227
automate.io [6]	Business	53
CloudWork [7]	Business	91

TABLE I: A list of trigger-action platforms, which we briefly surveyed, indicating their stated application area. Many of these platforms support integration with physical devices.

HTTP(S). These online services use the popular OAuth protocol to enforce authorization [31], [32]. Users must connect several such channels, before they can accomplish any useful work. Either the trigger-action platform developers or online service providers can implement channels. In the latter case, the trigger-action platform exposes a separate API to channel writers to help them integrate their online service with the platform. There is generally a one-to-one correspondence between online services and channels in the trigger-action platforms.

- **Trigger:** A channel may provide triggers, which are events that occur in the associated online service. "A file was uploaded to a cloud drive" or "smoke alarm is on" are examples of triggers. These triggers correspond to APIs in the trigger online service. The online services are REST (Representational State Transfer) services that use JSON or XML.
- **Action:** A channel may also provide actions. An action is a function (or set of functions) that exists in the API of the online service. Examples of actions include "turning on a connected oven" or "sending an SMS." In this paper, we collectively refer to a channel's triggers and actions as *operations*.
- **Rule:** Rules are at the core of the trigger-action user experience, and they are the core functionality that these platforms enable. A rule stitches together various channels to achieve useful automation. A typical rule has two pieces. The "If" piece represents a trigger or an event that occurs on an online service. The "Then" piece represents an action that should be executed on the online service. For example, "If there is a smoke alarm, then turn off my oven." This rule integrates the smoke alarm channel's "alarm is on" trigger with the oven channel's "turn off the oven" action. Some trigger-action platforms permit a single trigger and a single action (*e.g.,* IFTTT), some permit multiple triggering conditions and actions (*e.g.,* Zapier), if-then-else conditions (*e.g.,* Microsoft Flow), and even mathematical functions while combining triggering data (*e.g.,* Stringify).

A trigger-action platform takes the form of a cloud service that executes rules at large scale. For example, IFTTT currently supports 11 million users, 54 million rules, and 1 billion rule executions per month [29]. The cloud service provides accounts where users can create rules using a simple UI. All the platforms we surveyed also provide mobile apps that serve as an interface to the cloud service. Therefore, a trigger-action platform is technically a combination of a cloud service and a mobile app. For brevity, we refer to the cloud service of a platform as the trigger-action platform, unless stated otherwise.

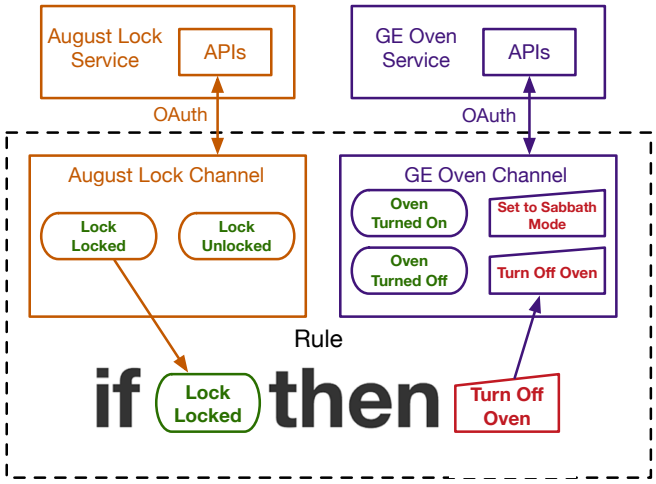

Fig. 1: Overview of trigger-action platform architecture in the context of a rule. Online services have a channel inside the platform. These channels gain access to online service APIs by acquiring an OAuth token during the channel connection step. A Rule combines a trigger and an action.

All of the platforms we surveyed use OAuth as the primary integration mechanism—this is expected as most of the online services today support OAuth based access for third parties. Our focus is on ensuring that a user's online services (and hence digital and physical resources connected to those online services) are protected from misuse. Therefore, we discuss the general authorization model of trigger-action platforms next.

Authorization Model. Online services protect their REST APIs using authorization protocols. OAuth is a popular choice that enables an online service to provide third parties with secure delegated access to its APIs. A trigger-action platform must obtain authorization to communicate with online services that its channels represent; and therefore must follow the OAuth authorization workflow. Figure 2 shows the four-step authorization model.

First, a channel developer (trigger-action platform developers or the online service provider itself) must create a client application for the online service's REST API. This client application represents a channel on the online service. During the sign-up phase, the online service assigns a client ID and a secret that the trigger-action platform uses during authorization.

Second, a user initiates a channel connection within the trigger-action platform administrative interface and this causes the platform to initiate the OAuth 2.0 authorization code flow—the recommended workflow for server-to-server authorization—that results in the platform requesting the corresponding online service for a short authorization code on behalf of the user. The platform passes a client identifier value, a redirect URI, and a scope value as part of the HTTP(S) request. The scope value represents the level of access the trigger-action platform is requesting to operate a channel. This authorization request results in the user being presented with an OAuth permissions screen that explains the scope that the platform is requesting. As the OAuth protocol does not specify the design of the permissions screen, the screen design, scope explanations, and UI options to modify the requested scopes is at the discretion of the online service.

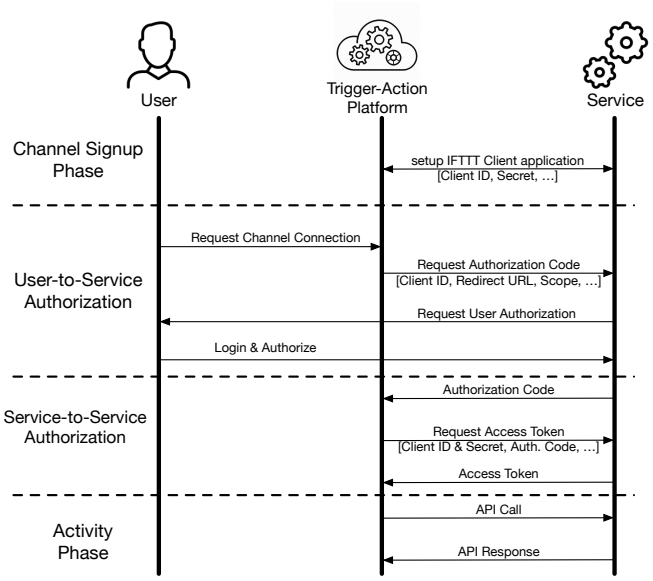

Fig. 2: The OAuth-based authorization model for trigger-action platforms has four phases. Channel developers create client applications for the online service that results in the online service assigning a client ID and secret to the application. Then, the trigger-action platform initiates an authorization workflow. The OAuth 2.0 authorization code flow is a popular choice, and it results in the platform gaining a scoped bearer token that authorizes a channel to invoke APIs on an online service. Users are prompted to approve scope requests during this process.

Third, assuming the user accepts the scope request, the online service redirects to the trigger-action-platform-provided redirect URI with a short authorization code as an argument.

Finally, the platform exchanges the authorization code, client ID, and client secret for an access token using server-to-server communication. The trigger-action platform then uses the OAuth bearer token to initiate API calls on the online service to implement channel functions.

Although OAuth 2.0 is by far the most popular authorization protocol in trigger-action platforms, there are online services that use OAuth 1.0a. OAuth 1 does not have explicit scoping as part of its authorization workflow, but offers a similar concept when a client application signs up for the online service's API. During this phase, the developer can choose scopes to enable. For example, Twitter uses OAuth 1.0a, and it provides a settings item that allows a developer to change the access level of the client application, and hence, to change the scope of any tokens issued in the future.

III. Security Implications of Trigger-Action Platforms

In this section, we discuss the security risks that trigger-action platforms pose to a user's digital and physical resources. We focus on risks that arise due to high-level design choices, and do not focus on low-level implementation errors that might enable these risks (*e.g.,* XSS, SQL Injection, OS bugs). The risks we focus on are due to a compromise of the trigger-action platform, and are due to OAuth deployment issues. We do not focus on a compromise of the online services of the users—such an attack is independent of any trigger-action platform. For example, if a user's Facebook account or Google Home account was hacked, then an attacker can manipulate data and devices independently of any trigger-action platform the user may be using.

A. Platform Compromise

Cloud services, including trigger-action platforms, can be compromised through bugs in its implementation or design, through social engineering attacks, or through a combination of these. As all of the platforms we discussed in §II are cloud services, an exploit could target any software in the web stack. Indeed, such compromises are common occurrences today. In 2015 alone, more than 700 million user records were exposed through 1673 data breaches [8]. Prominent examples of cloud service breaches include Target [16], US voters database [1], Dropbox [9], and the recent Google Docs OAuth-based phishing attack that compromised one million users [36]. Therefore, even well-designed cloud services are not immune to persistent and sophisticated threats.

Thus, we are concerned with the risks posed to users' digital and physical resources as a result of a trigger-action platform compromise. The main security mechanism guarding user resources is the set of OAuth tokens. These OAuth tokens are often long-lived.[2] An attacker who compromises the platform will be able to use the OAuth tokens to invoke operations on the users' resources arbitrarily.

Furthermore, trigger-action platforms today follow a logically monolithic design—a compromise of the platform implies that OAuth tokens for all users will be accessible to the attacker. Therefore, we conclude that the choice of standard OAuth tokens coupled with a logically monolithic design poses a large-scale security risk to users' digital and physical resources.

B. Risks from OAuth Token Compromise

Trigger-action platforms do try to limit the risks of misuse of OAuth tokens by constraining the set of operations available on the channels. For example, IFTTT does not expose the operation of deleting files on the Google Drive channel because deleting files is considered too risky (it can lead to accidental or malicious loss of all data on a user's Google drive). A research question is whether the OAuth tokens acquired by these services, if compromised, can be misused to perform risky operations. We call this the problem of overprivileged tokens. We performed a case study on the popular IFTTT trigger-action platform to study the overprivilege aspect. We note that it is not our aim to be exhaustive in our analysis of overprivilege, as it is a known problem in OAuth systems—for example, Chen *et al.* discuss OAuth issues in the context of mobile applications [21]. Rather, our goal here is to highlight the risks that still exist in trigger-action platforms, despite attempts to limit them by eliminating dangerous operations on their channels.

[2]Even if they are not, these tokens can be refreshed using information stored in the trigger-action platform.

Channel	Scope	Example Overprivileged APIs	Description
Google Drive	drive, user.info, userinfo.profile, feeds, feed, spreadsheets, documents	https://www.googleapis.com/drive/v3/files/file-id	Deletes a file
		https://www.googleapis.com/drive/v3/files/file-id/permissions	Creates a permission for a file
		https://www.googleapis.com/drive/v3/files/file-id/revisions/rev-id	Permanently deletes a revision of a file
Particle	ifttt	https://api.particle.io/v1/devices/device-id	Flashes a device with a pre-compiled binary
		https://api.particle.io/v1/devices/device-id	Unclaims a device
		https://api.particle.io/v1/devices/device-id WITH BODY name=new_name	Renames a device
MyFox Home Control	nil	https://api.myfox.me:443/v2/site/site-id/device/cam-id/camera/recording/stop	Stops camera recording
		https://api.myfox.me:443/v2/site/site-id/device/dev-id/heater/on	Sets heater to 'on' mode
		https://api.myfox.me:443/v2/site/site-id/device/dev-id/socket/on or /off	Turns a device on or off

TABLE II: Examples of overprivileged APIs that IFTTT channels can access. These APIs are not used in any triggers or actions. We shortened Google Drive scope names for brevity.

*1) **Case Study Procedure**:* The goal of our case study is to examine the use of overprivileged tokens in trigger-action platforms. We focused our case study on IFTTT [10] due to its popularity and integration with a wide variety of IoT, smart home devices, and online services. For our study, we selected channels which had their online service API open to developers. Studying these channels and comparing their triggers and actions with the capabilities of the OAuth token obtained by IFTTT allowed us to isolate API calls that are not used by any trigger or action, but are accessible using the OAuth token.

*2) **Case Study Results**:* Table II shows a summary of our case study results. We find that in all cases, the OAuth tokens that IFTTT possesses are overprivileged. The root cause for overprivilege arises from:

- Coarse-grained scopes: Online services were not designed to support only trigger-action platforms. They are designed to support the most general of use cases. Therefore, the OAuth scopes are often coarse-grained, and may not necessarily be fine-grained enough to support only the set of trigger and action functions for channels in the IFTTT platform. This problem is not unique to IFTTT. Rather, it is common to all trigger-action platforms.
- Balancing usability and security: The channel abstraction strikes a balance in the usability-security trade-off. Users must "connect" a channel to their account on IFTTT and this includes the user following an OAuth authorization flow. As this process is done once per channel, users do not have to perform OAuth authorization flows whenever they create a rule. Alternatively, users would have to perform an authorization flow for every rule they create if the channel abstract did not exist, leading to a drastic increase in the number of permission prompts. Although the channel design reduces the number of OAuth permission prompts to one per channel connection step, it does force IFTTT to request OAuth scopes that are powerful enough to execute all operations that it currently supports, and possibly even future-proof itself by requesting a coarser-grained set of scopes for operations that might be supported in the future. This problem is not specific to IFTTT. Rather, it is common to any trigger-action platform that supports the channel abstraction.

We provide more detail on the overprivilege in our case studies below:

Google Drive. Our API testing reveals that the Google Drive IFTTT channel has the ability to delete a user's files. We confirmed this behavior using a token with the same scope as what the Google Drive IFTTT channel requests. This can cause data loss if the corresponding token is stolen. We observe that the Google Drive channel requests multiple scopes. However, the OAuth prompt only provides the user with a binary choice of approving or denying the request.

Particle. Our API testing reveals that the Particle IFTTT channel has the ability to flash new firmware to a chip. We used a token with `scope=ifttt`, which is identical to what the Particle IFTTT channel requests, and reprogrammed a chip by simply using a REST API call. This can completely change the functionality of the Particle chips and cause a variety of security and safety issues if the corresponding token is stolen. We also observe that the Particle OAuth prompt only provides the user with a binary choice of either approving or denying the permission request.

MyFox Home Control. This channel can arm or disarm the MyFox security system. However, our API testing reveals that it has overprivileged access to the MyFox Home Control API that allows it to stop live video recording, turn on/off electric devices, and change the state of the heaters. This can result in security breaches, overheating and large utility bills if the corresponding token is stolen. We also observe that MyFox Home Control does not provide any kind of scoped access. This forces the channel to request complete access to the API. Furthermore, the MyFox Home Control OAuth prompt only provides a binary choice during authorization—either approve all requested permissions, or deny the request.

We conclude that the OAuth tokens that trigger-action platforms negotiate can be overprivileged: (1) online services

only provide a fixed set of scopes that can be incompatible with the channel operations of the platform, forcing it to request overprivileged access, (2) the usable channel abstraction necessitates tokens that can invoke multiple APIs in the online service, even if the user does not create rules that use all those APIs.

Therefore, trigger-action platforms pose a long-term security risk to users' digital and physical resources. An attacker who compromises the platform can misuse OAuth tokens to execute APIs arbitrarily, and can even invoke APIs outside the abilities of the trigger-action platform itself due to overprivilege.

IV. Towards Mitigating Risks of Trigger-Action Platforms

Our high-level goal is to develop a defense mechanism that mitigates the security risks outlined above. In this section, we first discuss our threat model, which we derived from our earlier analysis. We then explore candidate designs and highlight their shortcomings. Finally, we introduce the Decentralized Action Integrity concept, and discuss why it provides meaningful security guarantees for trigger-action platforms. We also discuss challenges in applying this concept to real platforms.

A. Threat Model

We adopt a strong but realistic attacker model—we assume that the trigger-action platform is untrusted, and can be compromised. An attacker can leak OAuth tokens, and then attempt to invoke actions arbitrarily. An attacker can also try to manipulate any triggering data passing through the platform. We assume that the online services of the user such as Facebook, Samsung SmartThings etc., are not compromised. If they are compromised, then an attacker can achieve its goals independently of the trigger-action platform.

The following aspects are outside our threat model. We do not prevent leakage of sensitive data (*e.g.,* the fact that a trigger has happened, or an attacker eavesdropping rule execution) from a compromised trigger-action platform (§VII contains a discussion of techniques to enable data confidentiality). We also do not prevent denial of service attacks.

B. Design Space Exploration

Under the above threat model, we discuss candidate designs to mitigate the security risks of a compromised trigger-action platform. We also highlight where these candidate designs fall short of providing necessary security and functionality properties. Our goal is to prevent attackers who have stolen the platform's OAuth tokens from arbitrarily invoking actions. We are concerned with actions because they have the ability to change the state of data and devices. In a physical setting, this can have dangerous physical consequences.

Short-Lived OAuth Tokens. One option is for online services to issue OAuth tokens that must be refreshed frequently. If the trigger-action platform is compromised, the online services can simply stop processing refresh requests from the trigger-action platform, and it can expire all issued tokens. This technique reduces the useful attack window to the refresh interval plus the time it takes for the knowledge that the platform was compromised to propagate to the online services. However, it relies on timely detection of the compromise. The strategy also depends on the existence of a separate signaling mechanism that the platform operator can use to contact the online services, as the platform itself is under the control of the attacker in the worst case.

Fine-Grained Tokens and Per-Rule Permission Prompts. If online services support very fine-grained tokens, trigger-action platforms could request tokens whenever a user programs a rule. Therefore, the trigger-action platform only has the amount of privilege necessary to execute rules. However, this increases the number of permission prompts for users, leading to usability issues. Additionally, attackers can still misuse fine-grained tokens. As we discuss later, our work improves on this basic approach by solving the misuse and usability issues.

Avoiding Bearer Tokens. Another solution is to use OAuth 1.0 tokens because these are not immediately useful to attackers if they are stolen in isolation. It requires stealing the signing key as well. However, if the trigger-action platform is compromised, then the attacker gains access to the signing key as well.

Fully Decentralized Platform Construction. A different approach would be to avoid amassing OAuth tokens in a cloud platform and provide trigger-action functionality to each user through a client that executes rules on their own machine (*e.g.,* mobile phone, voice-enabled assistant, or smart home hub for IoT scenarios). While this model removes the trigger-action platform as the single valuable target, it does not provide the benefits of cloud services such as fault tolerance, convenience, and availability.

Rule Analytics/Anomaly Detection. An analytics or anomaly detection system could potentially determine if certain operations are inconsistent with the set of user rules. Although this is a good defense-in-depth measure, it does not address the root of the problem—until a detection occurs, the attacker can cause harm. Furthermore, such systems typically require fine tuning of false positives and negative rates.

C. Decentralized Action Integrity

None of the above candidate designs prevent a compromised trigger-action platform from arbitrarily manipulating data and devices. In this section, we introduce the principle of Decentralized Action Integrity. A trigger-action platform that adheres to this principle drastically reduces the power of an attacker who compromises the cloud service component of a trigger-action platform: It prevents arbitrary misuse of action functions, and it prevents all users from being affected if the platform is compromised, even with leaked OAuth tokens. This concept manifests itself through the following four elements:

- **Rule-specific OAuth tokens**: If attackers obtain rule-specific tokens, they can only use the tokens to execute operations on the online services specific to rules that a user explicitly creates. For example, consider the rule "IF smoke is detected THEN turn off oven." The trigger-action platform would need two rule-specific OAuth tokens. One token allows it to only setup a callback for the smoke event. The second token allows it to only turn off the oven. Therefore, rule-specific tokens only allow the bearer to execute a REST APIs in the online service that are specific to a given rule. A rule-specific token can also constrain the arguments of an API

call. For example, it is possible to mint a rule-specific token that only allows the bearer to set a thermostat to 68 degrees Fahrenheit.

- **Timely and verifiable triggers**: The bearer of a rule-specific OAuth token can only execute an action function if it can prove that the corresponding triggering event was true within a reasonable (configurable) time period. Considering our example rule above, the bearer of the rule-specific token for the oven online service can only turn off the oven if it can prove to the oven service that the triggering event (smoke was detected) is true. If the platform attempts to turn off the oven without the proof of trigger occurrence, then the operation is denied.
- **Data integrity**: Any triggering data passing through the cloud service of the trigger-action platform should not be tampered with. Consider the rule "IF new NASA Instagram post, THEN save the picture to my Dropbox." It should not be possible for a compromised cloud service of a trigger-action platform to replace the Instagram image with malware.
- **Decentralized tokens**: Compromise of the cloud service of the platform should not imply that all tokens are leaked. There should not be a single point of failure.

The above four elements prevent an attacker with access to the OAuth tokens of a rule from arbitrarily invoking the action function, and they prevent a compromise of the platform from affecting all users of the platform. Rule-specific OAuth tokens combined with timely and verifiable triggers provide the security property of *action function misuse prevention and trigger misuse prevention* (§V-A).

Supporting Decentralized Action Integrity in a practical trigger-action platform requires overcoming several challenges. We discuss them next.

- Rule-specific tokens will drastically increase the number of OAuth permission prompts because the platform can only negotiate such tokens when the user is programming a rule. Connecting a channel to a user account will not involve negotiating any rule-specific tokens as the user is not programming any rules at channel connection time. The challenge is to maintain the same number of OAuth permission prompts while supporting rule-specific tokens.
- Verifying that a triggering event was true can introduce an undesirable dependency between the trigger and action service. In a naïve design, a way to support this proof would be to make the trigger service send an out-of-band signal to the action service whenever a trigger event occurs. This completely defeats the purpose of trigger-action platforms, which is to connect online services that have no common connections.

V. Decentralized Trigger-Action Platform Design

In this section, we discuss the design and implementation of Decentralized Trigger-Action Platform, a trigger-action platform that embodies Decentralized Action Integrity. We also discuss how our design overcomes the challenges discussed above. Our design introduces extensions to the OAuth protocol to ensure that a compromised trigger-action platform only has the necessary amount of privilege to execute the set of rules of a given user and that it cannot execute any actions that

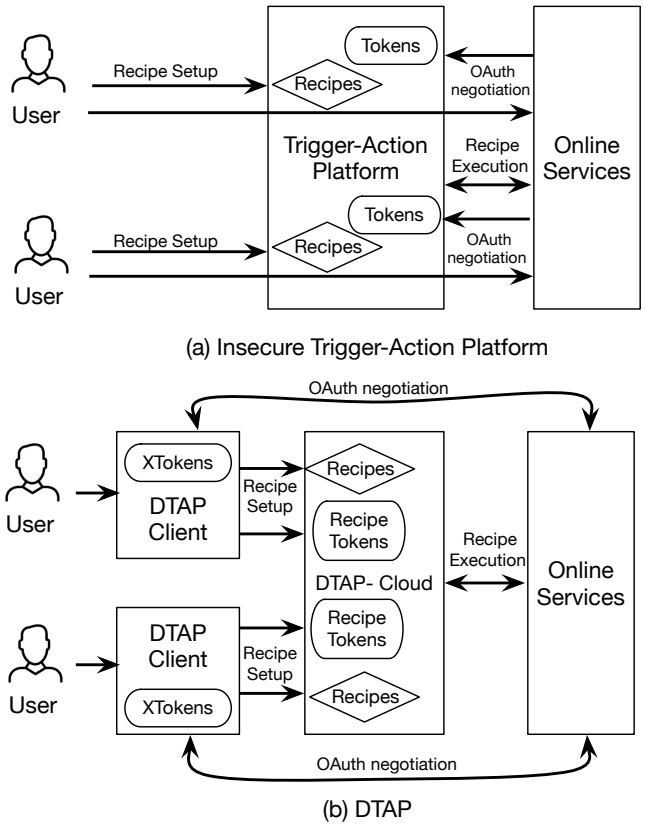

Fig. 3: High-level comparison between (a) insecure trigger-action platforms and (b) DTAP. Instead of storing the over-privileged tokens in the cloud, each user takes advantage of a DTAP-Client to secure his tokens. Only recipes and recipe-specific tokens reside in the DTAP-Cloud. DTAP guarantees that no other action other than the one specified in the recipes can be performed using the recipe-specific tokens. We note that the DTAP-Clients are *not* created or managed by the DTAP-Cloud. They are independent entities—a compromise of the DTAP-Cloud does not compromise a DTAP-Client.

are inconsistent with a user's rules. Where applicable, we cite IFTTT as a prototypical trigger-action platform.

Figure 3 provides a high-level comparison between Decentralized Trigger-Action Platform and current insecure designs. Decentralized Trigger-Action Platform splits the logically monolithic trigger-action architecture into a cloud service (DTAP-Cloud) that users do *not* trust, and one client per user, (DTAP-Client), that is trusted by *its user*. The DTAP-Cloud provides computational infrastructure to execute rules at large scale, similar to a trigger-action platform's cloud infrastructure. Each user must install a DTAP-Client on a device such as a smartphone. Users connect channels to their accounts and setup trigger-action rules with the help of the their clients. A user trusts its own client to manage highly-privileged access to their online services. The client can use hardware-backed secure storage (*e.g.,* secure keystore available on both iOS [19] and Android [34]) to ensure the security of these tokens.

Trust Model. We designed the OAuth protocol extensions for

DTAP to be open allowing anyone to implement the client portion of the protocol. Our design requires the clients to *not* be implemented by the same entity implementing the untrusted cloud service. Instead, we envision a community of developers building client applications and hosting them at various market places, *e.g.,* Android or Apple store. These app market models naturally result in a few well-built apps emerging, thus making it easy for users to install relatively good and secure implementations of the DTAP-Client. An example of such behavior exists with widely-used protocols—there are hundreds of SSH apps on Google Play, but only a handful have the highest ratings and installations. We see this kind of behavior for other protocols as well—Telnet, FTP, etc. We envision a similar model for the trusted clients. Additionally, the open source community can independently vet open source clients. Our design also requires the online services (*e.g.,* Facebook, Dropbox, SmartThings REST services) to not be compromised and to be trusted by the user. Finally, our design requires a method for an online service to cryptographically verify statements generated by another online service. The current prototype leverages the existing certificates (keypairs) in the public key infrastructure already in place for online services. However, this is merely a deployment convenience.[3] As we discuss in §VII, from the viewpoint of key re-use, a cleaner implementation is to use self-signed certificates, as one of the basic properties of our protocol is to ensure that a statement from an online service has not been forged or tampered with.

As is the case with a typical trigger-action platform, there are two phases to create a rule: Channel Connection, and Rule Setup.[4] We will discuss how these two phases work in our design, with the help of an example rule shown below. Without loss of generality, and for simplicity, this rule: (1) does not contain predicates in the condition. See §V-C for an explanation of how DTAP handles predicates securely; (2) contains a single trigger and single action. Although this is the most widely used and supported type of rule, there are trigger-action platforms that support multiple actions (including IFTTT) and sometimes multiple triggers. Handling extra triggers and actions does not affect our protocol—the trusted client and online service endpoints simply have to repeat the steps for a single-trigger single-action rule.

```
IF new_item added to ShoppingList THEN
   email new_item to x@y.com
```

Decentralized Trigger-Action Platform introduces two types of OAuth tokens:

- **Rule-Specific Token**: This token is fine-grained, and only allows the bearer to execute a single function with specific parameters on an online service. The DTAP-Client transmits this token to the untrusted DTAP-Cloud, where it is used to execute a user's rules. We introduce rule-specific tokens to limit the abilities of an attacker who steals them.
- **Transfer Token (XToken)**: This token is coarse-grained, and it permits the bearer to negotiate a rule-specific token *without* creating an OAuth permission prompt. It is never provided to the untrusted cloud platform and is only used by a DTAP-Client to acquire rule-specific tokens directly from an online service. When available, our design leverages a trusted-hardware-backed keystore to encrypt XToken storage when they are not in main memory (§VI). We introduce the notion of an XToken to maintain the usability experience of one-time authorizations of channels, and to gain the security of rule-specific tokens.

Channel Connection. In our system, a user connects channels using the user-specific client, typically running on user's smartphone or a trusted hub within the user's home. To create the above rule, the user first connects the ShoppingList and Email channels (assuming they haven't been connected before). This involves the usual step of the user logging in to the services corresponding to the channels with a username and password, and eventually accepting the OAuth scopes being requested. During the subsequent OAuth negotiation, the DTAP-Client requests and receives an *XToken*.

Rule Setup. Once the user has connected the trigger and action channels, the next step is to setup the trigger part of the rule. This involves navigating a UI and eventually clicking on one of the trigger functions that the channel offers (see Figure 5). The DTAP-Client retrieves a list of trigger and action names from the trigger and action services during channel connection, and then displays them in the UI. There is a one-to-one correspondence between trigger and action names displayed in the UI, and the actual scopes that are eventually requested by the DTAP-Client on behalf of the user. Therefore, the process of inferring the trigger or action scope the user intends on granting is straightforward.

For our example rule, `OnNewItem` is a trigger that fires whenever a new item is added to the user's shopping list. DTAP-Client will treat the physical act of the user clicking a specific trigger function in the trusted client UI as an implicit authorization for it to obtain a rule-specific token that can only execute `OnNewItem`. In this way, our design avoids introducing additional permission prompts even though it uses fine-grained tokens (see §IV-B). It transmits the XToken it obtained earlier to the trigger online service including information about the specific function for which it wants a rule-specific token. As a return value, the trigger service will also transmit its X509 certificate to the client and the rule-specific token (Figure 4).

Rule-specific token example. Assume that the ShoppingList service offers two functions that external parties may call: `test()`, and `OnNewItem(String URL)`. The XToken allows the bearer to obtain a rule-specific token for any of these supported functions. In our example rule, an external party only needs to call `OnNewItem` with a String value of "https://DTAP-cloud.com/new_item." Therefore, the client can obtain a rule-specific token scoped to only execute OnNewItem('https://DTAP-cloud.com/new_item'). That is, a scope in DTAP is equivalent to the name of a function in an online service.

Our design relies on two principles to overcome the challenge of an increased number of prompts while using fine-grained tokens:

[3]Our measurements of 297 IFTTT channels indicate that only 2 used HTTP, while all others used HTTPS.

[4]We discuss channel signup after these two steps even though Fig. 4 depicts signup before these two steps because signup is an activity performed by online service developers instead of users.

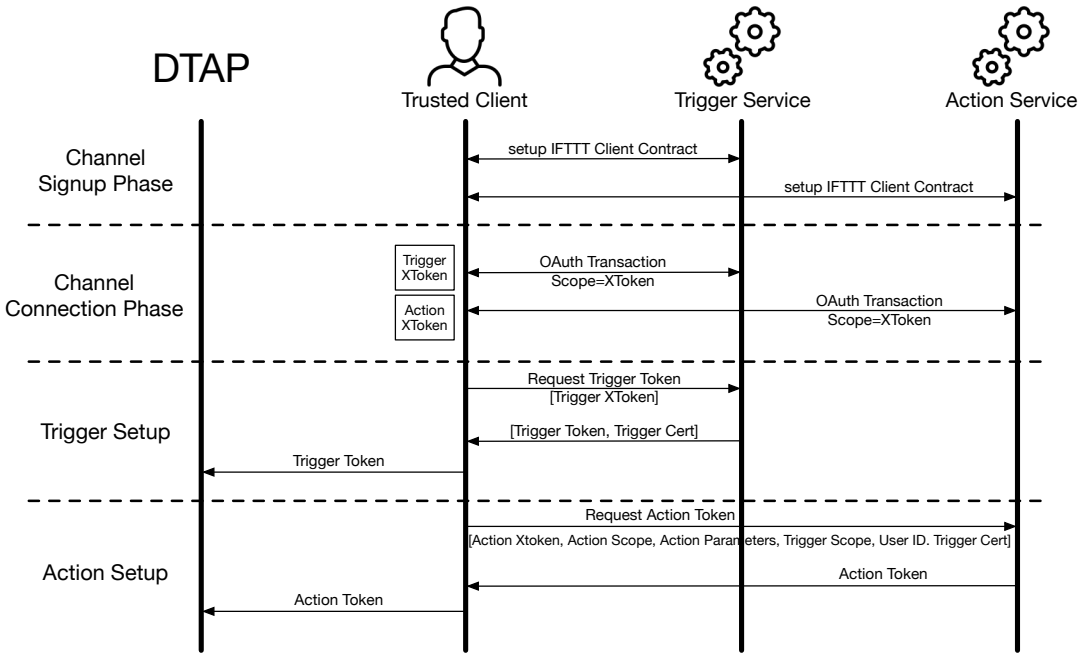

Fig. 4: DTAP authorization has four phases: Channel signup phase, where the clients obtain scope-to-function maps for every online service; channel connection phase, where the clients gain XTokens to online services the user wishes to use; and trigger and action setup phases where these tokens are used to request rule-specific tokens.

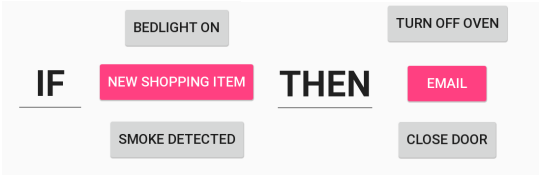

Fig. 5: Inferring user policy via user-driven access control in DTAP. The DTAP-Client enumerates possible triggers and actions. When a user clicks on a specific trigger and action, DTAP-Client automatically treats that as authorization from the user to negotiate rule-specific tokens. There is a one-to-one correspondence between trigger/action names and the scopes.

- The user authorizes the client to obtain an XToken when a channel is connected. This does not change the number of permission prompts for a user—it is the same as existing trigger-action platforms like IFTTT. The XToken has the property of allowing the client to obtain a rule-specific token without creating a permission prompt, as the user has already given the client that amount of privilege by authorizing it to obtain an XToken.
- The client only uses the XToken upon an explicit user interaction. This notion is inspired by User-Driven Access Control [38].

Setting up the action part of the rule is similar to setting up the trigger part. The user will navigate a UI and implicitly authorize the client to obtain a rule-specific token to invoke a particular action function. However, the token exchange process is slightly different. As Figure 4 shows, the DTAP-Client will transmit the action XToken, the trigger service's X509 certificate, the name of the trigger function (`OnNewItem`), the action function name (`send_email`), the triggering service user ID of the current user, and any action function parameters to the action service. The action service will return a rule-specific token and will associate all of this information with that token internally, effectively tying the issued token to a particular triggering function, a particular action function, and a particular user.

At this point, the DTAP-Client has obtained two rule-specific tokens needed to execute the rule. It transmits these tokens along with a description of the rule to the DTAP-Cloud that uses the trigger token to set up a callback to itself whenever the trigger condition (*i.e.,* new item added to shopping list) occurs.

We note that rule setup involves the trusted client, and therefore, depends on the availability of this client. Unlike in today's trigger-action platforms that host the rule setup interface as a highly available cloud service, our design permits rule programming only when a user client (such as a phone app or desktop app) is available. We do not view this as a significant shortcoming considering the rarity of rule setup and overall improvement in security. Furthermore, the most critical function of trigger-action platforms—executing rules at large scale—runs independent of the trusted client and therefore retains the benefits of cloud computing for high reliability and availability.

Channel Signup. Currently, a trigger-action platform knows which scopes to request for various trigger and action functions because channels store that scope-to-function mapping in the platform's cloud infrastructure. However, in our case, this

infrastructure is untrusted. DTAP-Cloud could manipulate scope-to-function mappings to trick the clients into requesting the wrong scopes. Our design solves this problem by requiring the online services to create a signed scope-to-function mapping and host those mappings at a well-known location. An online service signs its mapping using the private key corresponding to its X509 certificate. The clients retrieve these signed mappings during the channel signup phase (Figure 4).

Rule Execution. At runtime, whenever a new item is added to the shopping list, the trigger service will generate an HTTP call to the DTAP-Cloud and pass the trigger data (in our example rule, this will be the item that was added to the shopping list). DTAP changes this process slightly, and instead requires the trigger service to generate a trigger blob (see Figure 6):

$$TriggerBlob = [Time, TTL, TriggerScope, b64(TriggerData), UserID, SIG]$$

$$SIG = Sign\_with\_SHA256(TriggerServicePrivateKey, Time|TTL|TriggerScope|b64(TriggerData)|UserID)$$

The public key of the signing trigger service private key was transmitted to the action service as part of the setup process. $Time$ is the timestamp when the blob was created, and TTL specifies the period for which the blob is valid. Once the trigger service creates this blob, it will transmit it to the DTAP-Cloud. At that point, the DTAP-Cloud will lookup the appropriate rule, and then invoke the action function using the rule-specific token it obtained earlier. During this invocation, the DTAP-Cloud will make an HTTP request and include the trigger blob, the rule-specific action token, and parameters.

We note that during rule execution, neither the XTokens, nor the trusted clients are involved. Execution proceeds only with the rule-specific tokens. Therefore, our design retains the benefits of highly-available cloud services.

End-to-End Rule Verification. The DTAP-Cloud executes the rule by transmitting the trigger blob and the action token to the action service (Fig. 6). The action service will first execute a lightweight verification process before invoking the target function. The verification steps are:

V1: Verify that the passed rule-specific token exists.

V2: Verify the signature on the trigger blob using the X509 certificate of the triggering service. b64 denotes base 64 encoding. The action service does not need to interpret the format of the triggering data. It just needs to verify the signature over the entire triggering blob, which contains base 64 encoded trigger data.

V2.1: Ensure that $Time > PreviousTime$ where $Time$ is an extracted time stamp from the passed trigger blob, and $PreviousTime$ is the previously seen value for this triggering service. This prevents replay attacks from a compromised DTAP-Cloud if the attack occurs inside the TTL (see below). In practice, this requires reasonably precise timestamps.

V2.2: Verify that $TTL <= (Now - Time)$.

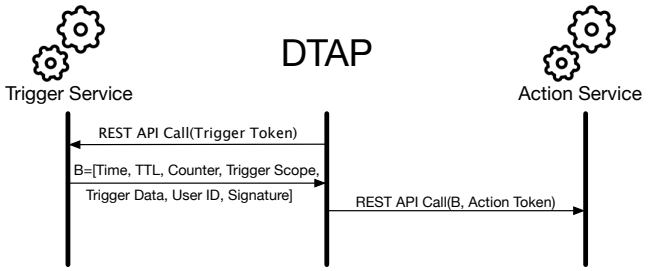

Fig. 6: Rule execution in DTAP: Upon a trigger activation, the trigger service contacts DTAP-Cloud with a trigger blob (B). DTAP-Cloud transmits this blob and the rule-specific action token to the action service. The trigger blob contains information the action service needs to verify that the corresponding trigger occurred for the specified user.

V2.3: Check that the trigger scope (function name) inside the blob matches what the action service was given during the setup phase.

V2.4: Verify that the UserID in the trigger blob is the same that was given to the action service during rule setup.

V3: Verify that the HTTP function being called at runtime is the same as the function name given by the trusted client to the action service during the setup phase.

V4: Finally, verify that the function parameters match those that the trusted client gave the action service during the setup phase.

If all verification checks succeed, then the action service proceeds normally and executes the `send_email` function. We note that the rule execution process does not depend on the DTAP-Client, as rule-specific tokens are already uploaded to the DTAP-Cloud.

A. Security Properties of DTAP

The above design ensures that the DTAP-Cloud can only execute user rules whenever a trigger occurs, even if it is compromised. Here, we explain in more detail how the various components of our OAuth protocol additions and decentralized design provide this guarantee.

Action Function Misuse Prevention. An untrusted or compromised trigger-action platform can invoke action functions at will, even in the absence of any triggers. Furthermore, based on our case study results, it could invoke a wide variety of functions given the overprivilege. DTAP prevents all of these problems. First, although XTokens are coarse-grained, they are never transmitted to the untrusted cloud service. Only rule-specific tokens that can execute a single function with specific parameters are transmitted to the DTAP-Cloud. Furthermore, through the signed trigger blob, the DTAP-Cloud can successfully execute an action function only if it can prove that a trigger occurred within some reasonable amount of time in the past.

Trigger Misuse Prevention. DTAP-Cloud could try to misuse the trigger blob and attempt replay attacks. The time stamp ([V2.1]: $Time > PrevTime$) and time-to-live value ([V2.2]: $TTL <= (Now - Time)$) ensures trigger blob freshness.

However, an attacker could in theory conduct replay attacks within the TTL period. In practice, this TTL is in the order of a few hundred milliseconds, and thus limits the ability of the attacker drastically. Additionally, the forward rolling timestamps also serve to further limit a replay attack ([V2.1]). The attacker could also try to use a trigger blob from another trigger service, the trigger blob of a different trigger function on the same service, or a trigger blob of a different user with the same rule. However, while setting up the rule, the trusted DTAP-Client instructs the action service to associate the name of the trigger scope (function name), and the user ID with the action token. Furthermore, the signed trigger blob contains this trigger scope ([V2.3]), and the user ID ([V2.4]). Therefore, DTAP-Cloud can only use a given trigger blob for a specific action function and for a specific user. In other words, the DTAP-Cloud can only execute the user's rule.

Trigger Data Integrity. The untrusted DTAP-Cloud may attempt to modify the data it receives from the triggering service before delivering it to the action service. An example of this would be a rule that saves new images from an Instagram channel to a Dropbox account. An attacker may replace the image with malware before uploading the file to Dropbox. DTAP protects against such an attack by requiring the trigger service to sign the fields of the trigger blob with its private key. When receiving the trigger blob, the action service verifies the signature using the public key that was associated with the action token during rule setup ([V2]).

Recipe Deletion. A user can delete rules with the help of DTAP-Client, that will issue a rule deletion HTTP API call to the online services involved in a specific rule. The online services will then invalidate the rule-specific tokens. A malicious DTAP-Cloud can retain the rule description, but it won't be able to execute any trigger or action functions because the online services will automatically refuse the HTTP calls as the tokens no longer exist.

No Single Point of Failure. Although the XToken is coarse-grained, it is never transmitted to the untrusted cloud service. The attacker has to target and compromise individual devices to obtain XTokens. Therefore, the cloud platform is no longer a single point of failure for the whole system. In §VII, we will discuss how DTAP can handle cases where the user has multiple trusted devices or if their device gets compromised.

B. End-User Properties of DTAP

From an end-user perspective, DTAP retains the concept of the one-time operation of users connecting channels to their accounts. However, as users have to use a client app, it limits their mobility (see §VII for options to increase mobility). DTAP does not add any additional OAuth prompts—it leverages User-Driven Access Control to automatically obtain the rule-specific tokens.

As we discussed in §III, online services in general neither provide users with fine-grained control over OAuth permissions, nor provide good descriptions of the permissions being requested. However, DTAP enables fine-grained control and good descriptiveness. When a DTAP-Client requests the user's permission to obtain an XToken, it directly lists the set of online service functions for which the XToken can be used to gain access. Furthermore, the online service can provide an option for users to select the set of functions they wish to include in the XToken—DTAP-Client will not be able to obtain rule-specific tokens for any functions not in that set.

C. Expressivity of DTAP

For services that do not natively support a callback interface for a specific triggering condition, the trigger-action platform must poll the service and check the triggering condition itself. For example, a weather channel might only offer an API that returns the current temperature. To support a trigger that fires if the temperature goes above 80 degrees, a trigger-action platform would poll the weather service and compute the predicate $currTemp > 80$. However, the DTAP-Cloud might simply ignore the result of the comparison, and invoke the action service repeatedly. The verification on the action end will succeed since DTAP-Cloud will obtain a valid signed trigger blob when it polls the trigger service.

DTAP handles such situations by allowing the client to associate a predicate with the action token. This predicate is expressed over fields of the trigger data part of the signed trigger blob. The DTAP-Client simply maps the condition the user sets up while creating the rule to a predicate and then instructs the action service to associate the predicate with the resulting rule-specific token. At runtime, the action service performs the additional step of verifying that the predicate is true.

Encoding such stateless predicates handles a significant fraction of the kinds of conditions that trigger-action platforms supports. We studied the triggers, actions, and online service APIs for 24 of the top channels in the IFTTT platform and did not find any predicates that required storing state. We also studied the Zapier channel creation process but did not find any resources for channels to keep state [18]. Moreover, all Zapier predicates only involve simple boolean operators. Our prototype fully supports expressing such rules.

D. Deployability of DTAP

We stress that our intent is to provide a starting point for the community to improve the security of trigger-action platforms from the ground up. Therefore, DTAP is a clean slate design. However, there are several components of DTAP that are readily deployable, and there are a few components that can be deployed with some effort. Below, we structure our discussion around the software components that DTAP design impacts.

- **OAuth 2.0 Protocol:** DTAP additions to OAuth are fully backward compatible on the wire. In order to provide Decentralized Action Integrity, no changes are required to existing software implementing the protocol. We conclude that the protocol additions are readily deployable.
- **Online Services:** DTAP does require changes to the online services that provide trigger and action APIs. Specifically, the online services need to understand rule-specific tokens and XTokens. As we will discuss in §VI, we have implemented a library that online services can use to gain the benefits of Decentralized Action Integrity. Developers would have to make one-line additions for each API call they want to protect using rule-specific tokens. Furthermore, to easen this transition process, in §VII, we

outline a proxy-based approach to incrementally introduce DTAP support in online services.

- **Trigger-Action Platforms:** DTAP does not require changes to the cloud component of the trigger-action platforms. We prototyped a version of DTAP that uses the existing Zapier trigger-action platform through its developer platform. We created custom channels that connect to prototype online services with DTAP support. We find that the existing cloud portion of the Zapier successfully transfers the trigger blobs from the trigger channel to the action channel and eventually to the action online service where the blob is subsequently verified. We conclude that trigger-action platforms can retain their existing rule execution and channel creation infrastructure while gaining the benefits of DTAP.

VI. Implementation & Evaluation

We implemented DTAP-Client on the Android platform. For additional client-side security, DTAP-Client will use a hardware-backed keystore, when available, to generate a key that we use to encrypt XTokens before storing them on the filesystem. Such keystores have been present in iOS devices since 2013 [19] and have been supported in Android devices since version 6.0 [34].

We built a Python library that online service developers can use to add DTAP functionality. The library provides a simple annotation (*i.e.,* Python decorator) that developers can place above sensitive HTTP API methods that require rule-specific scoping. The annotation automatically invokes the verification procedure (see §V). Using the Python library, we implemented the DTAP-Cloud, and two online services modeled after existing IFTTT channels: (1) an Amazon Alexa inspired ToDo list, (2) an email service. For our benchmark measurements, we implemented a skeleton version of IFTTT as our baseline. The skeleton version uses standard OAuth tokens.

A. Microbenchmarks

We first quantified micro-performance factors of DTAP. We created the following rule: *"IF new_item == 'buy soap' is added to MyToDo List THEN send_email(new_item)."* That is, if a new ToDo item with contents "buy soap" is added to the list, then send an email. This rule is representative of the kinds of rules that users can create on a typical trigger-action platform such as IFTTT. It contains all the elements of typical rules: a condition on data coming from the trigger service, and transfer of trigger service data to an action service function. We deployed DTAP locally, created the example rule, and then measured storage overhead, transmission overhead, and developer effort.[5] We found that using DTAP imposes negligible overhead: Each rule requires an additional $3.5KB$ in terms of storage, and an additional $7.5KB$ of transmission per execution. Online service developers using our prototype library only need to add a single line of code per HTTP API function—this is the same as that required by the popular oauthlib library for Python. We elaborate on the results below.

[5]For microbenchmarks, deployment location does not affect the quantities under study.

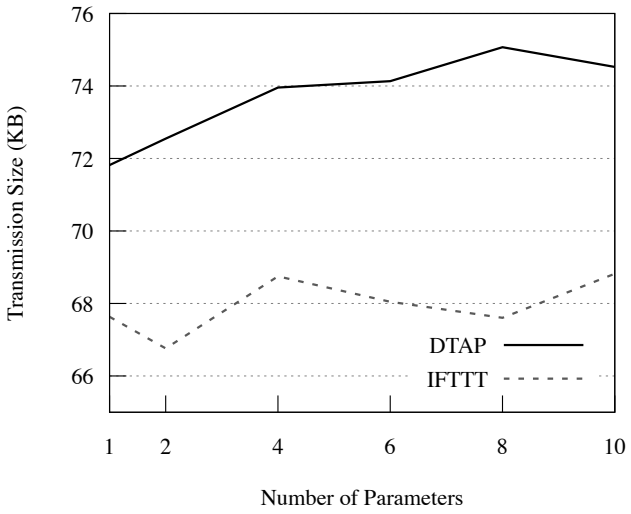

Fig. 7: Average total transmission size of baseline system and DTAP for $1-10$ parameters for 5 experiments. Although there is a linear increasing trend in both systems, the difference among the two remains negligible.

*1) **Storage Overhead**:* Using DTAP requires online services to store additional state: An online service needs to store an XToken for each trusted client that allows the client to create fine-grained tokens for individual rules. The online service also needs to store DTAP fine-grained tokens for each rule. These tokens include additional fields (*e.g.,* time, TTL), so they impose storage overhead on the online service. We computed the required storage for the baseline IFTTT system, and for DTAP. Our results show that each DTAP rule creates a $3.5KB$ overhead in addition to the $0.8KB$ required to store the XToken, compared to the $0.8KB$ storage cost for the baseline trigger-action platform. This extra token storage cost is negligible given the low price of storage and quantity of other user data that these systems collect.

*2) **Transmission Overhead**:* Executing a rule on DTAP requires transmitting more data over the network. This overhead is the result of additional data in the trigger blob (Figure 6) including time, TTL, and sign. To evaluate the transmission overhead, we computed the transmission size of rule execution in the baseline case and compared it to the same quantity in the DTAP case. We varied the number of function parameters passed $(1-10)$ and present the average result of five experiments. The number of function parameters matters because the rule-specific token information encodes data about the specific function being executed. We used Wireshark [17] to measure the flow sizes associated with ports assigned to online services and the DTAP-Cloud. Figure 7 presents this overhead for different number of function parameters for the two systems. In our experiments, DTAP created $6-11\%$ overhead. Even when using 10 parameters the transmission overhead does not exceed $7.5KB$. The variance in the results are due to normal network variances such as packet retransmission.

*3) **Developer Effort**:* We developed DTAP as a library for trigger and action services to make it easy for online service developers to transition to the DTAP model. Developers must only add a single additional line of code per function to protect it with DTAP verifications. When compared to existing OAuth

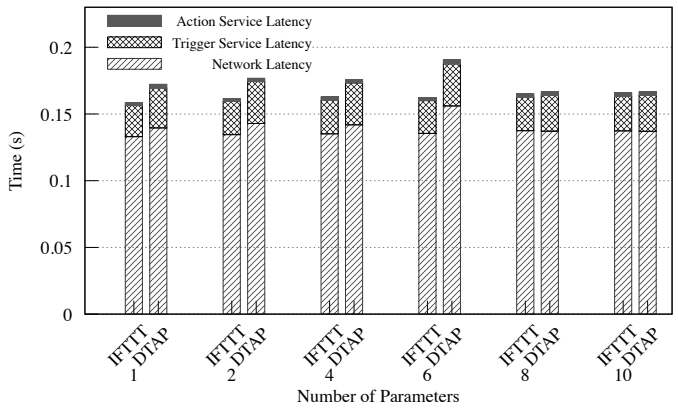

Fig. 8: DTAP adds less than $15ms$ of verification latency to rule execution compared to IFTTT.

libraries, such as the popular oauthlib [13], this is the same amount of effort—developers using oauthlib must also place a single annotation above HTTP API methods to create scopes.

B. Macrobenchmarks

We measured end-to-end latency and throughput of rule execution. We hosted the DTAP-Cloud and two online services on separate Amazon t2.micro EC2 instances. Each instance was configured with one 64-bit Intel Xeon Family vCPU@2.5 GHz, 1GB memory, 8GB SSD storage, Ubuntu 14.04 with Apache2, and MySQL Server 5.5. Our results show a modest $15ms$ latency increase, and 2.5% throughput drop in the online service when compared to the baseline (online service with no DTAP protections). This does not represent an inhibiting overhead for an online service especially when considering the effect of network latency and the lack of real-time requirements in these systems. We used the same ToDo list rule for our tests.

1) ***End-to-End Latency****:* We measured the time between the trigger service being activated due to an item being added to our ToDo list example rule, and the time the action service issues a `send_email` call. This time includes network latency, the time to generate a signed trigger blob, and the time to verify the trigger blob and the action token, in the case of DTAP. Our baseline case is a bare trigger-action system, and it only includes network latency, and time to execute the trigger and action functions without any DTAP verification. We varied the number of function parameters on the action service between 1 and 10. Figure 8 presents the results of these experiments. Our results show that excluding the network latency, the maximum verification overhead is less than $15ms$. For typical rules, that send emails, SMSs, or invoke actions on physical devices over a network, we consider this additional latency to be acceptable.

2) ***Throughput****:* We measure throughput as the number of rules executed per second, under a load of 2000 concurrent HTTP requests. We computed this concurrency level by examining the number of times the most popular IFTTT channel was used in rules (IF Notification channel was used in $1,514,188$ rules in our dataset). As per IFTTT's documentation, this channel will contact an online service once every 15 minutes [11], meaning that an online service would receive approximately $1,682$ $requests/second$. Therefore, we chose 2000 as an upper-bound for the number of concurrent requests a service would have to process. We used ApacheBench [3] to conduct throughput testing of DTAP and IFTTT by sending $10,000$ trigger activations with upto 2000 concurrent activations at a time. Table III presents our results, averaged over three separate runs. We find that DTAP decreases throughput by only 2.5%.

	DTAP		IFTTT	
	Avg	SD	Avg	SD
Throughput (req/sec)	94.03	8.48	96.46	5.74

TABLE III: DTAP reduces throughput by 2.5% compared to IFTTT. We used ApacheBench to send $10,000$ trigger activations with up to 2000 concurrent activations.

VII. Discussion and Limitations

Transitioning to DTAP. We discussed deployment options and compatibility issues of DTAP in §V-D. Although we facilitate migration to DTAP through development of an online library, direct transition from legacy system to DTAP might be still hard to achieve. One way to further ease this transition is the incremental addition of DTAP support to online services using a trusted proxy. Specifically, for each online service's REST API (set of function calls), we envision a trusted proxy running in front of it that intercepts OAuth and API calls, translates them from DTAP requests into regular requests, and vice-versa. Although this trusted proxy would be overprivileged, it does not increase the risk posed to the online service—an attack on the proxy is equivalent to an attack on the online service. As we stated in our threat model, if an online service is compromised, an attacker can achieve its goals independently of any specific trigger-action platform. The trusted proxy merely serves as an intermediate solution until the actual online service API is updated with DTAP support.

DTAP-Client use. In existing trigger-action platforms, users can login to the IFTTT website and create rules from any client device. However, DTAP requires users to create rules via a client device they trust (e.g., their smartphone), which stores XTokens in a private file system. Although our current client prototype does not support transferring client state from one device to another, building such functionality is fairly straightforward. One possible solution is to provide an export function to save the current client state to a disk image, and then provide an import function to load that client state into another device. If a client device is lost, then user rules continue to execute normally. However, the user will have to download the client again on another device, and go through the channel connection phase to re-establish the XTokens to create future rules.

A prototype limitation is that DTAP only allows a user to use a single trusted client at a time. The protocol itself does not preclude multiple clients, with users switching between devices running the trusted clients based on convenience (*e.g.,* a user at home may want to use a desktop to create rules, and the same user at the workplace may prefer to use a phone). Currently, our prototype does not include state maintenance between different clients of a single user. To support such a scenario, we envision the trusted client providing an option to back up the current state (XTokens, recipes, etc.) to a user's

private cloud storage (*e.g.,* Google Drive, Dropbox, etc.). The trusted client can automatically keep this private cloud state updated, and whenever the user logs in to a trusted client running on a different device, the client can download the state information securely.

Client-Device Loss. If a client device is lost, existing procedures to erase device data take care of removing OAuth tokens. Also, an "erasure-app" can be built to automatically contact online services and invalidate tokens with co-operation from our modified OAuth helper library. We leave implementing this to future work.

XToken Security. The XToken is a high-powered credential. Although DTAP reduces its vulnerability to leakage by design, a malicious client can still leak this credential. Such leakage however, only affects the single user and does not pose a risk for other users of the DTAP platform. As discussed, our implementation encrypts XTokens at rest using hardware-backed keystores when available. We envision further security by only performing XToken-related operations inside trusted execution environments (*e.g.,* Intel SGX on desktops, or ARM TrustZone on phones). We leave implementing this to future work.

Data confidentiality and Privacy. Our design currently reduces the privilege of the DTAP-Cloud—it only gains access to APIs and hence data it needs to run the user's rules. This is an improvement over the current state-of-the-art where we have shown through our analysis that an attacker can gain wide access to data and devices. However, even with our improvements, an attacker can still gain access to sensitive information simply by passively recording rule execution. A potential way to provide data confidentiality in this case is to encrypt data passing through the DTAP-Cloud. However, this can result in a loss of expressivity. Currently, trigger-action platforms can evaluate predicates on trigger data (see our weather data example in §V-C). Although the action service can solely evaluate these predicates, it does increase computational burden, thus defeating the purpose of trigger-action platforms. As our analysis shows, the predicates are stateless and involve simple comparison operators. Therefore, a potential solution is to leverage advancements in use-case-specific homomorphic encryption for secure integer comparison, rule matching, etc., to allow the least-privilege DTAP-Cloud to evaluate predicates on encrypted data [35], [43]. DTAP design also enables action services to collect data on triggers by recording the trigger scope and trigger certificate during rule execution. While the trigger service and the trusted client can obfuscate the trigger scope, a current limitation is that the action service can still profile obfuscated trigger names.

Self-Signed Certificates. Our current implementation reuses the HTTPS certificates of the online services. In order to avoid problems associated with key reuse, another implementation is to use self-signed certificates. A trigger service can generate a self-signed certificate and send that along with the XToken to the trusted client. The client can then send that certificate to the action service that associates it with the rest of the action token parameters. When a trigger occurs, the trigger service will use a private key corresponding to that self-signed certificate's public key to sign the trigger blob. At verification time, the action service uses the public key in the self-signed certificate to verify the blob. Such an implementation also avoids the privacy issues of revealing the identity of the trigger service to the action service, as the common name of the trigger service in a self-signed certificate can be obfuscated.

Formal Verification. The DTAP protocol has not been formally verified yet. Our future work plan includes using automated cryptographic protocol verification tools such as ProVerif [14] to verify the security guarantees.

VIII. Related Work

Trigger-Action Platform Studies. A few studies have investigated IFTTT in recent years, although in different contexts. Ur *et al.* [46] crawled the site in 2015, collecting 224,590 IFTTT programs shared by over 100,000 different users. Their study shows many interesting statistics including the number of different trigger and action channels used by IFTTT users. In contrast, our work investigates the long-term security risks that such platforms pose, and introduces the notion of Decentralized Action Integrity to counter these risks.

Surbatovich *et al.* analyzed user-created rules (recipes in IFTTT terminology) in the IFTTT platform using an information flow control approach [45]. Their focus was to determine the risks that users face due to errors either in rule creation, or due to inadvertent chaining of rules. In contrast, our work focuses on discovering and addressing the security design deficiencies of platforms like IFTTT. Addressing programming errors is an orthogonal research direction—the TrigGen tool, for example, aims to avoid errors caused by users incorrectly creating rules that have insufficient triggering conditions [37]. Our work is not focused on rule correctness.

Poirot finds vulnerabilities that occur due to abstraction discrepancies [33]. The authors apply Poirot to IFTTT and find a CSRF attack. In contrast, we assume that the trigger-action platform is compromised, and design a decentralized platform that ensures that an attacker cannot arbitrarily invoke actions. Instead, an attacker can only invoke actions if they can prove trigger occurrence.

Decentralized Trust Management. Blaze, Feigenbaum, and Lacy discussed PolicyMaker [20], an approach that exhibits locality of control in managing trust relationships. The DTAP protocol supports a similar decentralized verification of trigger-action rules. Furthermore, our notion of an OAuth token associated with information that constrains its use, and the notion of signed trigger blobs is inspired by the PolicyMaker approach of binding keys to predicates that determine the actions for which those keys are trusted.

Kerberos TGT. The notion of XTokens generating recipe-specific tokens bears similarity to Kerberos single sign-on protocol where "ticket granting tickets" are used to acquire "service tickets" to prove the user's identity to other services without prompting the user. Kerberos protocol, however, relies on a trusted server to hold TGT tokens and is typically used in scenarios where the user, trusted server, and connecting services are all within one umbrella corporation.

OAuth Security Analyses. Since the Open standard for Authorization (OAuth) debuted in 2007 [31], a number of studies discovered flaws in the protocol and the way the protocol was implemented in web sites [25], [12], [27], [28], [40], [41], [44], [48], [49], [50]. Nonetheless, the OAuth protocol is still

popular and it is now commonly used in mobile applications as well. Since the protocol was initially designed for web sites, some of the important details of the protocol was up to developers' interpretation when adapting OAuth to a mobile application. Recent work scrutinized implementations of OAuth in many Android mobile applications [21], [47], [42], showing that the majority of deployments were vulnerable [21], [47].

Our work is an addition to this growing list of work discovering vulnerabilities associated with deploying the OAuth protocol. However, our purpose in studying OAuth issues for trigger-action platforms is to determine how an incorrect deployment affects the security properties of trigger-action platforms. As discussed, our conclusion is similar to that of prior work—if the OAuth tokens are leaked, attackers are free to use them at will. In the context of trigger-action platforms, this means that tokens guarding access to digital and physical resources for millions of users are at risk of being stolen and misused.

Fett *et al.* conducted a formal security analysis of the OAuth 2.0 standard, and in the process, discovered new vulnerabilities [26]. They also propose fixes and prove the security of the protocol in an expressive web model. These contributions are orthogonal to ours and our work will benefit from their fixes to the OAuth protocol.

Cloud Platform Compromise. Beside vulnerabilities in OAuth implementation, other attacks on trigger-action platforms may also expose user data to attackers. Massive data leaks are commonplace. Equifax [2], Target [16], and US voters database [1] are some of the most recent examples of such high profile leaks. Our work introduces the first decentralized trigger-action platform design with the security property of only allowing the attacker who compromises the platform to execute specific user rules.

IoT Security. Fernandes *et al.* analyzed the security of SmartThings and conducted attacks using stolen OAuth tokens [23]. This underscores the need to secure IoT platforms that use OAuth tokens. We also introduce extensions to the OAuth protocol to support rule-specific tokens. Fernandes *et al.* also introduce flow tracking properties for IoT apps using FlowFence [24] focusing on confidentiality in a centralized setting. Our work focuses on action integrity in a decentralized setting.

IX. Conclusion

Trigger-Action platforms stitch together various online services to achieve useful automation. These platforms work by gaining privilege to access user data and devices in the form of OAuth tokens. However, the logically monolithic designs of current trigger-action platforms lead to a long-range large-scale security risk—if the platform is compromised, attackers can leak OAuth tokens for all the users of the platform, and then misuse those tokens to cause damage. Furthermore, with a case study of the IFTTT platform, we discovered that the OAuth tokens are overprivileged, allowing an attacker to cause even further damage. To systematically tackle this security risk, we introduced Decentralized Action Integrity, a security mechanism that provides the guarantee that even if the OAuth tokens of a trigger-action platform are stolen, the attacker cannot misuse the tokens. Instead, attackers can only invoke action services if they can prove that the triggering condition was true for a given rule. We designed and implemented Decentralized Trigger-Action Platform (DTAP), the first trigger-action platform that supports Decentralized Action Integrity. DTAP takes a decentralized approach to trigger-action platform design instead of the prevailing logically monolithic design that is currently in use. Our design introduces the notion of an XToken coupled with rule-specific tokens and a cryptographic extension to the OAuth 2.0 protocol. We implemented this design as a Python library that can be easily integrated into existing services (one-line addition). Performance tests indicate modest overhead ($15ms$ latency increase; 2.5% throughput drop).

Acknowledgements

We thank the anonymous reviewers and our shepherd, Limin Jia, for their insightful feedback on our work. We also thank Tadayoshi Kohno for providing valuable feedback. This material is based in part upon work supported by the National Science Foundation under Grant No. 1646392 and 1740897, by the UW Tech Policy Lab, and by the MacArthur Foundation.

References

[1] *191 Million US Voter Registration Records Leaked In Mystery Database*, http://www.forbes.com/sites/thomasbrewster/2015/12/28/us-voter-database-leak/.

[2] *A Brief History of Equifax Security Fails*, https://www.forbes.com/sites/thomasbrewster/2017/09/08/equifax-data-breach-history/#3829d1f4677c.

[3] *ApacheBench*, http://httpd.apache.org/docs/2.4/programs/ab.html.

[4] *Apiant*, https://apiant.com/.

[5] *Automate Processes + Tasks— Microsoft Flow*, https://flow.microsoft.com/en-us/.

[6] *automate.io*, https://automate.io/ifttt-alternative.

[7] *CloudWork*, https://cloudwork.com/.

[8] *Data Breaches Exposed 707 Million Records During 2015*, http://news.softpedia.com/news/data-breaches-exposed-707-million-records-during-2015-501116.shtml.

[9] *Hack Brief: 4-Year-Old Dropbox Hack Exposed 68 Million Peoples Data*, https://www.wired.com/2016/08/hack-brief-four-year-old-dropbox-hack-exposed-68-million-peoples-data/.

[10] *If This Then That*, https://ifttt.com/.

[11] *IFTTT- Learn More*, https://ifttt.com/wtf.

[12] *OAuth Security Advisory: 2009.1*, https://oauth.net/advisories/2009-1/.

[13] *oauthlib 2.0.0*, https://pypi.python.org/pypi/oauthlib.

[14] *ProVerif*, http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[15] *Stringify*, https://www.stringify.com/.

[16] *Target Expects 148 Million Loss from Data Breach*, http://time.com/3086359/target-data-breach-loss/.

[17] *Wireshark*, https://www.wireshark.org/.

[18] *Zapier*, https://zapier.com/.

[19] Apple Inc., "iOS Security - iOS 9.3 or later," 2016.

[20] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, ser. SP '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 164–. [Online]. Available: http://dl.acm.org/citation.cfm?id=525080.884248

[21] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 892–903. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660323

[22] G. Cooking, *If your smoke alarm detects an emergency, then turn off your oven*, http://tinyurl.com/gv4q3hq.

[23] E. Fernandes, J. Jung, and A. Prakash, "Security Analysis of Emerging Smart Home Applications," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[24] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "FlowFence: Practical Data Protection for Emerging IoT Application Frameworks," in *Proceedings of the 25th USENIX Security Symposium*, 2016.

[25] D. Fett, R. Küsters, and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[26] D. Fett, R. Küsters, and G. Schmitz, "A comprehensive formal security analysis of oauth 2.0," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1204–1215. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978385

[27] E. Homakov, *How we hacked Facebook with OAuth2 and Chrome bugs*, http://homakov.blogspot.ca/2013/02/hacking-facebook-with-oauth2-and-chrome.html.

[28] ——, *OAuth1, OAuth2, OAuth...?*, http://homakov.blogspot.ca/2013/03/oauth1-oauth2-oauth.html.

[29] IFTTT, *IFTTT Platform Size Metrics*, https://platform.ifttt.com/pricing.

[30] *IFTTT*, https://ifttt.com/.

[31] Internet Engineering Task Force, *RFC5849 - The OAuth 1.0 Protocol*, 2010.

[32] ——, *RFC6749 - The OAuth 2.0 Authorization Framework*, 2012.

[33] E. Kang, A. Milicevic, and D. Jackson, "Multi-representational security analysis," in *Proceedings of the 2016 ACM International Symposium on the Foundations of Software Engineering*, ser. FSE '16, 2016.

[34] *Hardware-backed Keystore*, https://source.android.com/security/keystore/.

[35] R. P. Kim Laine, Hao Chen, "Simple Encrypted Arithmetic Library - SEAL (v2.1)," Tech. Rep., September 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/simple-encrypted-arithmetic-library-seal-v2-1/

[36] M. Mimoso, *1 million Gmail users impacted by Google Docs phishing attack*, https://threatpost.com/1-million-gmail-users-impacted-by-google-docs-phishing-attack/125436/.

[37] C. Nandi and M. D. Ernst, "Automatic trigger generation for rule-based smart homes," in *PLAS 2016: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, Vienna, Austria, October 24, 2016.

[38] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 224–238. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.24

[39] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984. [Online]. Available: http://doi.acm.org/10.1145/357401.357402

[40] B. Security, *How I Hacked Any Facebook Account...again!*, http://www.breaksec.com/?p=5753.

[41] ——, *How I Hacked Facebook OAuth to Get Full Permission on Any Facebook Account (Without App "Allow" Interaction)*, http://www.breaksec.com/?p=5734.

[42] M. Shehab and F. Mohsen, "Towards Enhancing the Security of OAuth Implementations in Smart Phones," in *International Conference on Mobile Services*, 2014.

[43] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 213–226. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787502

[44] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems," in *CCS*, 2012.

[45] 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 *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 1501–1510. [Online]. Available: https://doi.org/10.1145/3038912.3052709

[46] B. Ur, M. P. Y. Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman, "Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes," in *CHI*, 2016.

[47] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu, "Vulnerability Assessment of OAuth Implementations in Android Applications," in *ACSAC*, 2015.

[48] R. Wang, S. Chen, and X. Wang, "Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services," in *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[49] R. Wang, X. Wang, L. Xing, and S. Chen, "Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation," in *CCS*, 2013.

[50] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization," in *USENIX Security*, 2014.

[51] R. Yang, W. C. Lau, and T. Liu, "Signing into one billion mobile app accounts effortlessly with oauth2.0," in *BlackHat*, 2016.