Architecting Trigger-Action Platforms for Security, Performance and Functionality

Deepak Sirone Jegan
University of Wisconsin-Madison
dsirone@cs.wisc.edu

Michael Swift
University of Wisconsin-Madison
swift@cs.wisc.edu

Earlence Fernandes
University of California San Diego
efernandes@ucsd.edu

***Abstract*—A Trigger-action platform (TAP) is a type of distributed system that allows end-users to create programs that stitch their web-based services together to achieve useful automation. For example, a program can be triggered when a new spreadsheet row is added, it can compute on that data and invoke an action, such as sending a message on Slack. Current TAP architectures require users to place complete trust in their secure operation. Experience has shown that unconditional trust in cloud services is unwarranted — an attacker who compromises the TAP cloud service will gain access to sensitive data and devices for millions of users. In this work, we re-architect TAPs so that users have to place minimal trust in the cloud. Specifically, we design and implement TAPDance, a TAP that guarantees confidentiality and integrity of program execution in the presence of an untrustworthy TAP service. We utilize RISC-V Keystone enclaves to enable these security guarantees while minimizing the trusted software and hardware base. Performance results indicate that TAPDance outperforms a baseline TAP implementation using Node.js with 32% lower latency and 33% higher throughput on average.**

I. Introduction

Trigger-action platforms (TAPs) enable end-users to automate interactions between a wide variety of third-party online services and devices [50]. For example, an end-user can create an applet that is *triggered* when a new row is added to a Google spreadsheet; the applet performs some transformation on that row data and then sends an *action* to another service, such as posting a notification on Slack. Popular TAPs include IFTTT [10], Zapier [65], and Microsoft Power Automate [51].

As TAPs are large-scale systems with millions of users [50], they become centralized hubs with privileged access to user data and devices (e.g., the popular IFTTT platform boasts 20 million users [44]). Their current design requires users to place full trust in their secure operation. Specifically, users must trust that (i) the TAPs only access the minimum necessary data for running applets, (ii) the TAPs faithfully execute applets without modification, and (iii) the TAPs keep the access tokens secure from misuse and breaches.

All of this trust is unwarranted, and as we will show, unnecessary as well. TAPs are essentially cloud services, and thus, they are vulnerable to all security and privacy issues that plague cloud services [4], [6], [18], [19]. For example, an attacker could exploit a bug in the web stack to steal OAuth tokens and then use them to access sensitive data or an attacker could compromise parts of the cloud service to violate integrity of applet execution. Beyond external attackers, TAPs themselves can access sensitive data and make it available to unrelated parties [42]. Consequently, some third-party services (e.g., GMail) have become reluctant to interface with TAPs citing privacy concerns [43]. At the same time, users are becoming wary of the insufficient safeguarding of their data by TAPs [36]. In an ideal world, a TAP would only execute user-created applets while ensuring that attackers cannot manipulate them or steal their data.

A growing line of work in TAP security is exploring alternative designs for trigger-action platforms with the goal of achieving approximations of this ideal world [19], [23], [28], [32]–[34], [38], [55]. These works explore different points in the design space and have various trade-offs among security, performance and functionality under a threat model that the TAP itself is untrusted. They provide some subset of the following security guarantees: (1) applet execution integrity — an applet executes without tampering; (2) applet data confidentiality or minimization — sensitive user data is either not accessible to attackers or only accessible in "least privilege" form; (3) trigger freshness and replay protection — triggers cannot be delayed without detection and the products of applet execution cannot be replayed multiple times to the action service. These works use the threat model of an untrustworthy TAP because it is a convenient proxy for a variety of real threats such as leakage of access tokens, vulnerabilities in the web stack, malicious insiders with privileged access, etc.

We contribute to this line of work by designing, implementing and evaluating TAPDance, an alternative TAP architecture under the same threat model as prior work (i.e., the TAP is untrusted and cannot guarantee the security properties above). TAPDance achieves a better trade-off between security, performance and functionality compared to prior work (Section IX contains a comparative analysis). Specifically, our work achieves better performance and functionality than eTAP [33] and Walnut [55] while offering similar security (under different assumptions), and stronger security than minTAP [32], DTAP [38] and oTAP [34]. We achieve these desirable points in the design space because of *tailored* use of trusted execution environments (TEEs).

The straightforward use of trusted execution environments would place the entire TAP runtime into an enclave. This requires running a TypeScript interpreter and any support libraries, leading to a large trusted software and hardware computing base. For example, the enclave would have a complex interaction with the untrusted operating system outside, increasing the probability of various attacks. This situation is no different from the status quo — any bug or vulnerability that was present in current TAP systems is simply transported to the enclave environment. An attacker who exploits these issues will gain access to the enclave, leading to the same

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23034
www.ndss-symposium.org

security problems we have with current TAPs.

Therefore, a key challenge and consequently, primary contribution of our work is determining how to re-architect TAPs so that a minimal amount of software runs inside the attested enclave with a small set of trusted hardware primitives supporting that isolated execution environment. Our insight is that we can model applets as pure computations. That is, an applet is a pure function that receives trigger data (e.g., spreadsheet row), transforms that data to compute an action (e.g., a message for Slack). Thus, the ideal design is to run this pure computation inside an attested enclave while keeping the rest of the untrusted TAP infrastructure outside, and therefore, isolated from the user's sensitive data and computation. In this model, the user only trusts their applet code and a hardware root-of-trust embedded in the datacenter processor. This core insight leads to a system design that offers a better trade-off between security, performance and functionality compared to prior work on TAP security.

TAPDance achieves this ideal design by addressing several challenges. First, we want TAPDance to support real user applets written in the TypeScript interpreted language. This provides better functionality (i.e., the types of support applets) than systems like eTAP [33] or Walnut [55]. At minimum, it would require running the TypeScript interpreter inside the enclave, leading to a large trusted software computing base. Instead, we run machine code corresponding to applets inside enclaves. We create a compiler for a restrictive subset of the TypeScript language that we design based on our decision to model applets as pure functions. As we show in Section VII-A, this design yields a smaller TCB relative to the straightforward approach of running a TypeScript runtime inside an enclave. We also show that this approach is sufficient to express and execute a large fraction of real-world applets (642/682 applets in our evaluation).

The second challenge is that there are a variety of trusted execution technologies with different security and functionality trade-offs [21], [39], [46], [48]. Theoretically, TAPDance could run on any of these technologies. In keeping with our design principle of minimizing trust, we desire the simplest possible trusted execution environment that meets our needs. We synthesize a set of TAP-specific security requirements and map them to various trusted execution technologies (Section IV-B). We find that RISC-V enclaves (e.g., Keystone [48]) best fit our needs because they support a simple hardware mechanism for isolating contiguous and small chunks of private memory suitable for running small compiled applets (i.e., physical memory base/bounds protection registers). RISC-V enclaves also offer customizability — a property that is important in addressing the next challenge.

The final challenge is to ensure freshness on applet execution. Freshness of data is not a standard primitive offered by TEEs. An applet should only execute once in response to fresh triggering data. The straightforward solution is to implement nonces inside applets. As explained later, this is problematic because it would require the trigger service to become aware of applets and it would also require applets to wait while new trigger data is being fetched. TAPDance avoids these issues by offering a centralized nonce management service as part of the enclave environment. The customizability of Keystone RISC-V enclaves allows us to integrate this into the security monitor.

Contributions.

- We re-architect trigger-action platforms to balance security of user programs, performance at scale and functionality. Specifically, we co-design the applet abstraction and trusted execution environments to support applet data confidentiality, integrity and execution freshness while supporting real user code with a lower performance penalty compared to existing approaches. A core insight is to model user programs as pure computations. This opens up a design space that allows for tailored and efficient use of enclave technology.
- We implement and evaluate TAPDance, a trigger-action platform that enforces the above security properties using RISC-V enclaves. TAPDance has a 5.2x reduction in the software TCB needed for running applets. Based on running real user programs, TAPDance outperforms a baseline Node.js TAP system (which does not run inside an enclave, mimicking current TAP architectures) with a 32% lower latency and 33% higher throughput on average. We see a performance improvement because TAPDance enclaves run machine code of programs instead of interpreted TypeScript. We also show that our pure computation abstraction of user applets is expressive enough to run a majority of real code (642/682 applets from the minTAP dataset).

Designing least-privileged computer systems requires a tailored approach that leverages the properties of the application domain. We show how trusted execution environments can be tailored to enforce least-privilege on trigger-action platforms in the sense that the platform may only execute user-created programs without doing anything else, such as reading confidential data and sending it to third-parties or modifying user programs to perform actions beyond what the user intended.

All of our code is open source and available on Zenodo [1].

II. Background

Trigger-Action Platforms (TAPs). A TAP is a cloud service that connects different web services and enables end-users to build useful automation through a simple trigger-compute-action paradigm. The web services range from digital resources like Dropbox, Gmail, Google Calendar and Slack to physical resources like IoT devices (e.g., door locks, lights). Web services expose *triggers* that notify the TAP about an event (e.g., "calendar event about to start" or "door unlocked") and also expose *actions* that allow the TAP to issue operations (e.g., "send a message" or "turn on the light"). For each trigger and action, the services host APIs to handle the communication with the TAP.

Trigger APIs supply trigger data objects to the TAP that contain various properties, such as `Title`, `Starts`, `Ends`, and `Description` in the context of our example

[1] https://zenodo.org/record/8287340

rule in Figure 1. A user creates an *applet*, a small TypeScript program that receives trigger data, transforms it and creates an action object. The TAP uses the action object to invoke an action API. Applets can depend on multiple trigger data sources and can issue multiple actions. Without loss of generality, we focus on the case of a single trigger and action in the rest of the paper and point out how our approach extends to multiple triggers/actions in Section VIII.

For interoperability with third-party services, popular TAPs like IFTTT and Zapier specify a compatibility or shim layer that the participating services must implement to host TAP-specific APIs and translate the service's original authorization and data APIs into a format that the TAP understands [45], [66]. We utilize this shim layer in our work and modify its behavior to support our security guarantees.

In summary, TAPs are large-scale distributed systems. They support running applets for millions of users interacting with hundreds of trigger/action services. Their current design requires users to place complete trust in the correct and secure operation. Experience with large-scale cloud services has taught us that this trust is unwarranted because of the lack of proper privacy controls in companies and the presence of exploitable bugs in complex software codebases [3], [14], [40].

RISC-V Enclaves with Keystone. At a high level, enclaves provide an attested and isolated execution environment that guarantees confidentiality of data and integrity of code in the presence of malicious supervisor-mode software. Intel SGX or AMD SeV offer fixed enclave designs incorporating memory encryption. We use RISC-V enclaves in our work, along with Keystone, which is a framework that uses RISC-V memory protection primitives to build customizable enclaves [48]. Section IV-B provides further rationale choosing RISC-V.

Keystone manages enclaves using a small trusted machine-mode software called the *Security Monitor (SM)*. The SM manipulates the RISC-V physical memory protection registers defining the base and bounds of isolated memory regions for enclaves. Each enclave has a code-identity, defined as a hash of its code signed by the processor's private key and the hash of the SM, also signed in the same way. The SM provides in-enclave services, such as the ability for an enclave to request a measurement of its code-identity. Enclaves can store data using keys bound to its code identity. Remote users can verify attestations of enclave code-identity to gain confidence that the code running remotely has not been tampered with.

Customizability in software aids our design goal of minimizing the trusted hardware base. We extend the SM's behavior to provide additional enclave services necessary for TAPDance security guarantees. We refer the reader to the Keystone paper for additional details on its operation [48].

```
if (GoogleCalendar.eventFromSearchStarts.Title
    .indexOf('IFTTT') === -1) {
    Slack.postToChannel.skip()
} else {
  Slack.postToChannel.setMessage('Now: ' +
  ↪  GoogleCalendar.eventFromSearchStarts.Title)
}
```

Figure 1: Example applet.

III. Design Considerations

Our goal is to ensure confidentiality and integrity of TAP applets against a malicious cloud infrastructure while minimizing the TCB in software and hardware. We discuss the threat model, security and functionality goals and outline design challenges, including an analysis of alternative approaches.

A. Threat Model

The TAP is a single entity that is entrusted with the privileged OAuth tokens for millions of users, making it a ripe target for online attackers. Real-world attacks continue to demonstrate that OAuth tokens can be stolen [15], [17] Each applet represents a strict, controlled and authorized use of the user's OAuth tokens. Violating the integrity and confidentiality of the applet execution is equivalent to the TAP getting arbitrary control over the trigger data and action services of all users. Additionally, prior TAP work has shown that OAuth tokens are over-privileged [32], [38], amplifying the risks. A recent line of work studies the security of trigger-action platforms under the assumption that it is untrustworthy [32], [33], [38]. We adopt the same threat model because it is a convenient proxy for real threats such as application-level security vulnerabilities in the web stack or cloud software stack on which a TAP runs. This work is concerned with remote attackers only: we assume that the attacker does not have physical access to the machines in the datacenter running the TAP software, ruling out physical attacks on the hardware. An attacker: (1) can attempt to arbitrarily manipulate applet binaries; (2) has control over the operating system running the TAP software; (3) knows API details of trigger and action services; (4) has access to OAuth tokens for trigger and action services of the user; (5) can modify, replay or drop messages between parties.

We also assume that a malicious user can create an account on the TAP service and submit arbitrary applet binaries with the goal of compromising the TAP and accessing other user data and code. An honest user would not intentionally attempt to leak their own data, but a user-created trigger-action program could be buggy. There is mutual distrust between the applets of different users and the TAP.

A user interacts with the trigger-action platform using an app on their client device (e.g., smartphone or laptop). Following prior work, we assume this client device is trusted and forms a root-of-trust for users of TAPDance [32], [33], [38].

We assume the trigger and action services are trusted (i.e., the services on which users have accounts like Google Calendar, Slack, GMail, Samsung, etc). They follow specified protocols and do not collude to weaken a user's security. Finally, we assume that the processor package running TAP-Dance is secure and fully trusted.

B. Security Goals

Our primary goal is to ensure the confidentiality and integrity of applet execution. Concretely, we aim to provide the following: (1) Trigger data and action data (resulting from applet execution) should never be revealed to the untrustworthy TAP or to another malicious user of the system; (2) Applets

should execute without tampering from malicious parties; (3) Applet execution should only occur in response to a fresh triggering event.

In addition, the untrusted TAP should not be able to abuse the OAuth tokens for the trigger and action services even with access to them.

Non-goals. Denial of service, network traffic analysis and network side channels are out of scope for this work. For example, an attacker could infer the semantic purpose of an applet just by inspecting the triggers and actions involved.

C. Functionality Goals

While ensuring the above security goals, we want TAP-Dance to be practical, and thus, we aim to provide the following functionality properties: (1) Support existing trigger-action applets written in TypeScript; (2) Maintain the current process of programming and deploying applets; (3) Trusted client device that help users create and deploy applets should not be required during applet execution; (4) Modifications to trigger and action services should be minimal and kept local to the TAP-compatibility layers, thus facilitating easier adoption. We setup these goals to retain the characteristics of trigger-action platforms that have made them popular among users and trigger/action services.

D. Alternative Approaches

Computation at the Edge. The trigger or action service could directly run the applet. This reduces the TAP's role to be a simple connector of services. However, this requires the trigger or action service to support an execution infrastructure similar to AWS lambda, significantly increasing the complexity of REST servers and exposing them to security issues from running untrusted third-party applets. Our goal is to retain the original role of the TAP – a cloud service that can run applets at large scale without changing the semantics of the endpoint REST services.

Cryptographic Approaches. A recent line of work uses cryptography to protect sensitive user data as it passes through the TAP. For example, OTAP encrypts data end-to-end but does not support applets [34]. eTAP also does end-to-end encryption, but can support applet computation using garbled circuits [33]. However, systems like eTAP incur high overhead, particularly on the user's client device because it has to generate a new garbled circuit for every trigger event.

IV. TAPDance Design

Our goal is to minimize the trusted software and hardware computing base while executing real trigger-action applets with confidentiality and integrity. We achieve this goal by using trusted execution environments because they offer the best trade-off in terms of performance and security compared to the alternative approaches outlined above. In this section, we discuss the design challenges in achieving this goal and then introduce TAPDance's design that achieves the security guarantees outlined in the prior section while overcoming the challenges.

"Big Enclave" Approach. To illustrate the system design challenges, we outline a simple "big enclave" approach and point out why its not a good design. In this approach, we run the entire trigger-action platform inside an enclave. This requires the enclave to support: (1) TypeScript interpreter because users write applets in that language, (2) System call support for that interpreter, (3) TLS and TCP/IP stack to communicate with trigger/action services, (4) UI stack that runs from an enclave so that users can safely login to their trigger/action services to provide OAuth tokens to the TAP, (5) Paging and memory encryption support (to enable swapping) because the enclave is large. Every applet execution will require creating an enclave with all of the above components (except the UI stack that is only required when a user provides access tokens to the system or creates a new applet).

The "big enclave" approach illustrates our main challenge — the trusted computing base in software and hardware is large, and any bugs in this large stack increase the probability that an attacker can compromise the enclave. The enclave also needs extensive untrusted OS support to service system calls, exposing it to Iago attacks [31]. Theoretically, one could limit this specific attack surface, but it would require expanding the enclave trusted code base even more to include a library operating system [25]. As stated, a core design goal is to minimize the trusted computing base while supporting real applets.

However, the "big enclave" approach also highlights a key security benefit of using TEEs — enclave isolation, and therefore applet isolation, is hardware-assisted. We note that isolation has three interpretations: (1) isolation between enclaves (2) fault isolation of an untrusted enclave from the OS and (3) inverse sandboxing protecting the enclave from an untrusted OS. Our design choice of using a TEE enforces all three interpretations of isolation using hardware assistance and a small privileged security monitor. This is a stronger security property than current TAPs that rely on JavaScript language-based isolation that has a history of bugs [19] and only provides a weak version of the first two interpretations of isolation above.

Finally, a major challenge of the TAP environment is that applets are short - much shorter than serverless functions, for example - and run infrequently. This requires an incredibly light-weight runtime environment. Standard use of TEEs with a full OS environment [11], [25], [30], [53], [57] to support unmodified POSIX applications would be an order of magnitude larger (several millions of lines of code), preventing storing them in memory, requiring much longer measurements (hashing of enclave pages) during startup and a larger interface with the host OS.

A. Challenges and Solutions

Supporting Real Applets. To execute real-world applets, we need a way to support TypeScript inside the enclave. However, as explained in the "big enclave" approach above, running a full interpreter is not a good idea because it leads to a large and complex hardware/software TCB. Instead, we observe that applets are small snippets of code that run on trigger data to produce action data and they do not need all of the power that comes with a TypeScript interpreter. Motivated by this, our insight is that we can model applets as pure computations that need few supporting libraries and language features to execute.

TAPDance Requirement	Hypervisor + TPM	SGX	AMD SeV-SNP/Intel TDX	TrustZone	Keystone
`Small Contiguous Isolated Memory Segments without Paging`	No	No	No	No	**Yes**
`Only Remote Attack Threat Model`	Yes	No	No	**Yes**	**Yes**
`Small SW TCB`	No	**Yes**	No	No (OP-TEE), **Yes (Komodo)**	**Yes**
`Small HW TCB`	**Yes**	No	No	**Yes**	**Yes**
`Secure Hardware Time Source`	**Yes**	No	No	**Yes**	**Yes**
`Secure Randomness in Hardware`	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
`Monitor Support for Freshness`	**Yes**	No	No	**Yes**	**Yes**

Table I: Applet security requirements compared against various TEE-like systems. Keystone TEE based on RISC-V best fits our design requirements.

As we show in Section VII, this model supports the majority of real world applets from the widely-used IFTTT platform.

To reduce the amount of runtime code inside an enclave, we choose to compile applet code rather than running a full interpreter. Thus, the applet executes as a binary with its minimal support libraries inside the enclave.

Combined with our insight of modeling applets as pure functions, we derive that TAPDance enclaves need small contiguous memory segments isolated from supervisor code — a single segment can contain applet code, a small stack and heap. Specifically, we do not need privileged code and hardware support for paging and memory encryption. Because applets are small and run to completion as pure functions, allocation is simpler: we can allocate segments of contiguous memory without suffering from fragmentation, and do not need to page memory to storage, which requires encryption and page-based allocation in the untrusted OS.

Compiling TypeScript to machine code is a complex undertaking, made difficult by the lack of type information and the presence of in-built features like the *eval()* function [29]. However, when applets are pure functions they only require a small subset of TypeScript that does not allow I/O, *eval()*, monkey patching or other complex TypeScript features.[2] Additionally, the trigger and action data types representing inputs and outputs in applets are fully known at compile time, making type inference simple. Table II captures the subset of TypeScript that we support in detail.

We admit this compiler into the TCB, but it is not a runtime component that the attacker can manipulate. In fact, a malicious applet writer can directly upload hand-coded machine instructions, but the fault isolation of the TEE will protect the rest of the system and other applets as well. The compiler could also generate vulnerable applet code, however, fault isolation protects the system and other applets. An attacker could send malformed trigger data in an attempt to compromise an applet, but, we assume that the trigger service is trustworthy and not under the attacker's control (see the threat model). Protecting trigger services is an orthogonal problem.

Programming Applets, Verifying Attestations and Trust Model. The user cannot trust the TAP to create applets, because the entire TAP software is untrusted. Our solution borrows a trust-reduction idea from prior work [32], [33], [38]. Specifically, each user trusts their client device (e.g., smartphone, browser) and it manages credentials for the user's accounts on trigger and action services. We assume this client device's hardware and operating system is secure. Users create and compile their applets on this trusted device, encrypt the applet and support data and then transfer it to the TAP, where the applet may only be decrypted inside an attested enclave. This reduces trust — each user trusts only their device (and the datacenter processor manufacturer) unlike today where users have to trust everything. While this solves the primary challenge, it introduces another — the user's client is not online and available every time an applet needs to run. Thus, what entity will verify the attestation on an enclave in which an applet is about to run? Our solution relies on a long-running "manager" enclave that helps the user establish trust in the applet enclave transitively.

Freshness Guarantees. One of our security goals is to ensure that applets only execute on fresh trigger data and actions run in response to fresh applet execution (i.e., replays are prevented). Freshness of data is not a standard primitive offered by TEEs. Secure time and a secure source of randomness are necessary components. A key challenge is that a single trigger may be consumed by multiple applets. If applets manage nonces on their own, they each require their own nonce that the trigger service must echo. In addition, if applets generate nonces they must start execution before contacting the trigger service and remain resident in memory during the communication delay. Our solution is to provide centralized nonce management in the security monitor that allows the untrusted operating system to request a fresh nonce and multiple applets to securely use the same nonce.

Limiting OAuth Token Misuse. To minimize the enclave TCB, we have to run the OAuth negotiation and token usage steps outside the enclave while ensuring that the untrusted TAP may only use those tokens in ways that are consistent with the user's applet. Our solution is to encrypt any trigger/action data using keys only known to the enclave code and service endpoints. Therefore, if the attacker tries to query the trigger, they will get an encrypted response that can only be decrypted inside a valid attested enclave. They could use the action service OAuth token to initiate an API call, but the action service will only accept a valid encryption that can only result from execution of an attested applet.

[2]Popular TAPs like IFTTT place similar restrictions on applets as well that prevent them from using TypeScript features such as I/O, monkey patching or the eval() function.

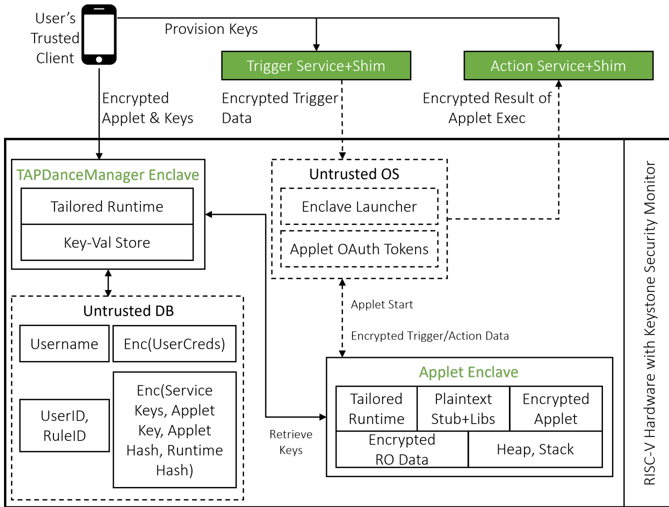

Figure 2: The TAPDance Architecture: Based on Keystone enclaves using RISC-V primitives.

B. TAPDance Components

The core design principle in TAPDance is to run the pure computation of an applet inside an attested enclave while keeping everything else outside. The trigger and action services only send/receive end-to-end encrypted data using keys only known to them and to the applet enclave. Only a valid attested applet enclave will get access to keys that will allow it to decrypt trigger data, compute on it, and produce a validly encrypted packet that the action service will accept. Fig. 2 shows a high-level design. We build TAPDance on top of the Keystone enclave that uses RISC-V primitives [48].

The choice to use RISC-V enclaves. In principle, any TEE can create an attested and isolated execution environment for applets. However, from a security perspective, we want to select a TEE that best fits our design needs while minimizing the software and hardware TCB. The prior section derived a set of applet security requirements that we summarize in Table I. The table also lists the extent to which a particular TEE technology meets an applet security requirement. Based on this analysis, we conclude that the Keystone TEE best fits our needs. In what follows, we will explain how we arrived at this best fit.

As stated, one of TAPDance's core insights is to model applets as pure computations that need minimal support libraries. Therefore, the TEE should provide the ability to create small contiguous memory regions that are isolated from privileged software. The RISC-V instruction set architecture supports physical memory protection (PMP) registers that provide exactly this service. Keystone is a TEE system built on top of the PMP mechanism and provides a simple security monitor that manages isolated memory regions. By contrast, other TEEs can also create small isolated regions, however, they depend on paging and optionally, page encryption. This brings additional complexity in software and hardware, leading to the potential for security vulnerabilities. Additionally, depending on the implementation, paging of enclave memory can lead to controlled-channel attacks [64], a problem that does not affect TAPDance because it uses PMP registers and does not need untrusted OS-managed paging. From a threat model perspective, TEEs like Intel SGX assume a physical attacker who inspects DRAM or the memory bus. This necessitates a page encryption scheme that cannot be turned off. By contrast, Keystone accomodates different threat models, including a remote-only attacker setting.

Additionally, Keystone offers customizability of the security monitor that runs in a special machine-mode privilege level. This allows us to include nonce management and offer it as a service to the enclave environment. Other TEEs may offer similar customizability, but that comes at the cost of complexity in software and hardware, as discussed above. Thus, we conclude that the Keystone TEE represents the best trade-off among all design requirements.

Trusted Client. In current TAP design, the user performs OAuth token negotiation and applet programming directly on the TAP. However, this is incompatible with our threat model where the TAP is untrusted. Therefore, we borrow from prior work in TAP security and use a trusted client device that serves as the user's root of trust — they only trust their device to interact with the TAP [32], [33], [38]. This device will help users: (1) negotiate OAuth tokens for the trigger/action services and provision them on the untrusted TAP cloud service; (2) create, compile and transmit applet binaries to the TAP; (3) establish secret keys with the trigger/action endpoint services and with the `TAPDanceManager` enclave.

Security Monitor. This is the standard Keystone Security Monitor that creates enclaves using the PMP registers on a RISC-V chip. TAPDance extends the monitor to offer four additional services to enclaves. First, it provides a secure time source and guarantees that the applet executes without preemption after fetching time. Second, it provides a secure source of randomness so applets can generate fresh nonces. Third, it implements a nonce management service. The untrusted OS can invoke the SM to generate a new nonce value, and applets can invoke the SM to verify that a nonce is fresh. Once an applet has verified a specific nonce, the SM will report the nonce is stale if it is presented again. Other applets, though, can still verify the nonce. This ensures *at most once* semantics for processing trigger data in an applet. Finally, the SM terminates applets that have run for too long.

`TAPDanceManager` Enclave. This is a long-running enclave whose code-identity is baked into the trusted client. It solves the challenge of verifying the attestation on an applet enclave while the user's trusted device is not online/available. The user's trusted client sets up an attested TLS connection to the `TAPDanceManager` to transmit the encrypted applet's key material. We use attested TLS to simultaneously verify the attestation on the `TAPDanceManager` enclave and setup a TLS connection. The user trusts that this enclave will correctly verify the attestation on the applet enclave when the time comes to run user-created code. The code of `TAPDanceManager` is assumed to be publicly available with reproducible builds that match the code-identity in the user's trusted client. The `TAPDanceManager` also securely stores key material that it needs to decrypt a user's applet code, using a key-value store backed using a sealing key tied to its code identity. Section V will discuss the protocol steps involving `TAPDanceManager` in more detail.

Untrusted TAP OS. This component manages OAuth tokens, receives encrypted triggering data for user applets, launches applet enclaves, pauses/destroys them, sends out encrypted action data and provides a networking stack. Our core security guarantees are designed to tolerate interference by this untrusted OS. Although the untrusted OS can use the OAuth tokens to issue API calls on the trigger/action services, the services either send encrypted data or will only act on encrypted data. Thus, the OS cannot misuse OAuth tokens. It can also attempt to replay trigger data to violate the applet execution freshness guarantee. In TAPDance, we let the untrusted OS/TAP fetch the encrypted trigger data once (via TLS) and place it in an untrusted buffer corresponding to all the applet enclaves that require the particular event's data before triggering the applets. Although replay from the trigger service is not a concern, previously fetched data by the untrusted OS/TAP can be replayed to each of the applet enclaves. TLS alone does not protect against replay by the untrusted OS/TAP when delivering data to the applet enclave in our design. Section V discusses how TAPDance protects against this using a series of timestamps and nonces with help from the security monitor.

Applet Enclave. User-created computations run as binary code inside applet enclaves. The untrusted OS allocates a new applet enclave and transfers control to its *decryption stub*. This stub will report the enclave's code identity to the `TAPDanceManager` enclave, who verifies the attestation and, if successful, supplies decryption keys over a secure connection back to the applet enclave. At that point, the applet enclave decrypts the user-specific code and executes. This process solves the attestation challenge and does not require the user's trusted client to be online or available.

Decrypted applets run pure computations supported by a tailored runtime that we provide. This minimal runtime offers services like JSON parsing, math functions, date and time parsing without requiring system call support. We also include a minimal TLS library so that applets can exchange data with the `TAPDanceManager` enclave over an attested TLS connection. When the applet completes execution, it gives the encrypted and integrity protected action data to the untrusted TAP for delivery to the action service.

Time-keeping. TAPDance runs an attested time-keeping enclave that uses NTP for clock synchronization. The security monitor is in charge of interacting with the time-keeping enclave and updating the processor's time registers. The trigger and action services use NTP as well, leading to a system where all clocks are loosely synchronized.

Trigger/Action Shims. Commercial trigger-action systems like IFTTT require endpoint services to implement shims to make themselves TAP-compatible. TAPDance requires trigger and action services to add minimal additional functionality to these shims to support the security guarantees. Specifically, the trigger shim will encrypt data using a user- and service-specific key before responding to a data request from the TAP. It will also manage a set of nonces in an event queue to ensure freshness of applet execution (Section V-C3 has more details). The action service uses its user- and service-specific key to decrypt data it receives from the untrusted TAP.

TypeScript-to-RISC-V Compiler. In TAPDance, the applets are pure functions that may only be written in a restricted subset of TypeScript. In addition to the restrictions placed by popular TAPs such as IFTTT [16], we document our additional restrictions in Table II. Our compiler supports a TypeScript variant similar to AssemblyScript [8], a restricted version of JavaScript that supports static compilation to WebAssembly. Inspired by this, TAPDance requires that all variables are statically typed at declaration time. We add support for built-in TypeScript methods on the supported types. As we show in Section VII, this subset is sufficient to express a significant majority of real-world IFTTT applets (642/682 are supported without any applet modifications). We built our compiler on top of StaticScript [13]. The support libraries include functions for accessing trigger and action objects and functions for running attested TLS so that the applet code can communicate with the `TAPDanceManager`.

Category	Restrictions
Types	Only primitive types, arrays, TAP-relevant trigger/action objects No unions, anonymous functions
Functionality	No monkey patching, *eval()*, file/network
Libraries	Pre-defined functions for accessing trigger/action data, secure system time, attested TLS, JSON parsing, math

Table II: TypeScript subset that we compile in TAPDance. All variables are statically typed.

V. Creating and Running Applets with TAPDance

We will discuss the various protocols involved in creating and running TAPDance applets. We structure the discussion around the applet lifecycle phases: (1) user bootstrapping, (2) applet creation, (3) applet execution. First, we discuss two primitives in the design for communication between various parties.

OCalls. Keystone enclaves provide an *OCall* as a mechanism to pass data to the untrusted OS, requesting a particular service. The enclave requests the SM to copy the function identifier of the requested service as well as the data for the particular function into an untrusted buffer shared between the untrusted OS and the enclave. The SM then notifies the untrusted OS of the pending OCall in the untrusted buffer. The untrusted OS executes the OCall corresponding to the function id, copies the return value to the untrusted buffer and notifies the SM, which notifies the enclave of the returned data. In TAPDance, the applet enclave and the TAPDance Manager enclave use the various OCalls listed in Table III.

Attested TLS. All communication between trusted parties occurs over attested TLS [61]. This is an extension to the TLS protocol so that it embeds verification of enclave reports during TLS connection setup. The TLS server certificate includes a custom extension which contains evidence that the certificate was generated at runtime inside an enclave with a particular code identity on a trusted processor. In TAPDance, this evidence is the server enclave's report with the additional data being the public key on the certificate. Recall that the additional data along with the enclave's hash is signed using the SM's private key to generate the signature on the enclave's

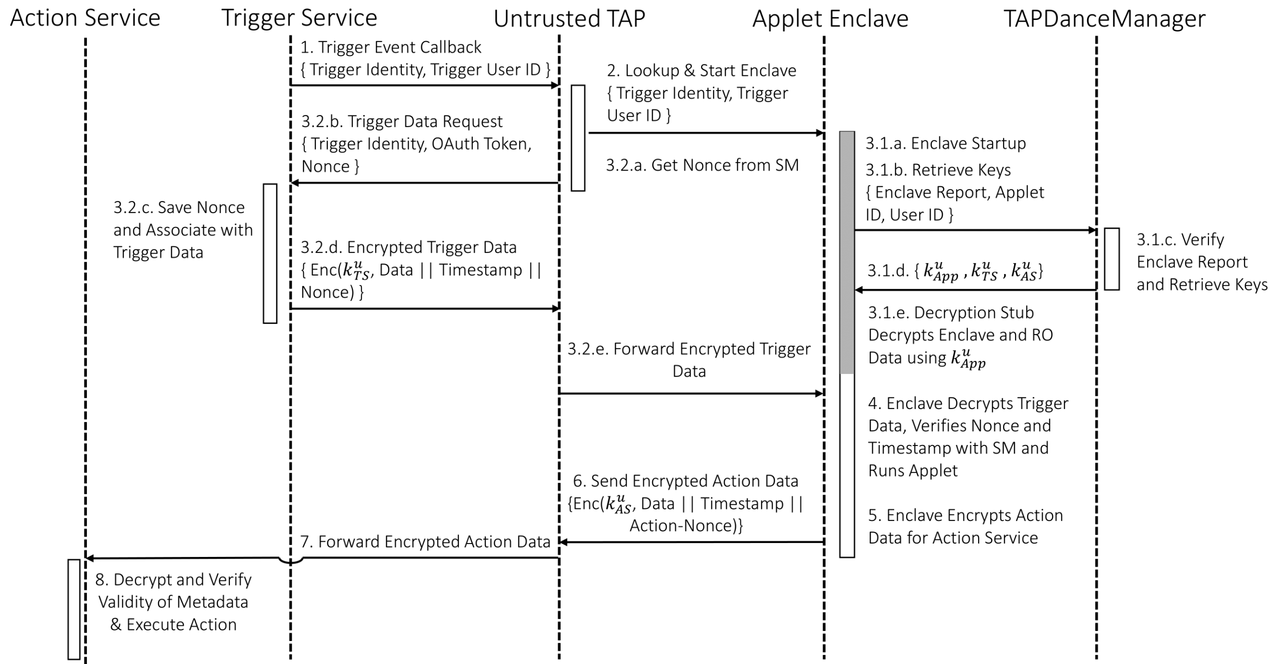

Figure 3: The TAPDance Execution Protocol. Sensitive data and results are encrypted everywhere except inside an attested applet enclave and the trusted endpoint services. Steps 3.1 and 3.2 occur in parallel, where the alphabetic identifiers indicate serial steps within those parallel events. The shaded region represents the additional steps that take place on a cold-start.

report. As the SM's private key is derived using the processor's secret key and also because only the server enclave can request its own report, the evidence proves that the server enclave has a particular code identity and it is running in a RISC-V processor with a particular secret key. The client verifies the enclave report on the certificate extension by (i) verifying the enclave and SM hashes (ii) verifying the signature on the enclave report that includes the public key on the certificate and (iii) verifying the signature on the SM's report using the processor's public key.[3]

A. *User Bootstrapping*

The user interacts with TAPDance through their trusted client. This smartphone app or browser extension embeds the code-identity of the `TAPDanceManager` enclave and the public key of the RISC-V processor package on which the `TAPDanceManager` is running. When the user signs up to a trigger or action service, the trusted client will generate and transmit user-specific symmetric encryption keys to these endpoint services (k^u_{TS}, k^u_{AS}). The user will also create a username and password on the TAP by interacting with the `TAPDanceManager` enclave over an attested TLS connection [61]. The `TAPDanceManager` stores the user's credentials in untrusted storage encrypted using a sealing key that is bound to its code identity. The appendix shows protocol details for user registration and login to the `TAPDanceManager`.

[3]The manufacturer issues a certificate of the processor's public key.

B. *Creating an Applet*

Using the trusted client, the user programs an applet by selecting a trigger event, optionally writing TypeScript code that transforms the trigger data and then selects an action. The client program compiles this applet to RISC-V assembly using our LLVM-based compiler. The trusted client then encrypts the applet binary using an AES GCM key k^u_{App}. It also attaches a decryption stub to the applet binary that is not encrypted. This decryption stub will help run the applet binary (discussed in the next section). The trusted client transmits the assembled applet package to the untrusted TAP over a standard TLS connection. It will also transmit k^u_{App}, k^u_{TS}, k^u_{AS} and the applet code-identity over an attested TLS connection to the `TAPDanceManager`, that stores this information using sealed storage.

The trusted client also negotiates OAuth tokens for the endpoint services and transmits them to the untrusted TAP. These tokens help the TAP execute applets by contacting the trigger service for data and sending the action service the results of applet execution. Recall that the untrusted TAP cannot misuse these tokens because the trigger service only responds with encrypted data under k^u_{TS} and the action service will only accept an API call if the data is encrypted under k^u_{AS}.

C. *Running an Applet*

Figure 3 shows the applet execution protocol. The untrusted TAP operating system is in charge of creating and running applets in RISC-V enclaves. Each applet is associated with a trigger identity, a string that is unique to the user, trigger

service and the trigger event attributes. The untrusted TAP operating system listens for event notifications on a generic API endpoint. When the trigger event occurs (Step 1 in Figure 3), the trigger service will send an HTTP(S) callback message to the untrusted TAP with this trigger identity. The trigger identity will help the TAP to efficiently lookup all applets listening on this event and then launch them (Step 2; trigger identity does not have a security purpose).

1) Applet Launch: The untrusted TAP launches an applet by first allocating a set of contiguous pages, loading the applet and runtime binaries in memory, setting up the page tables for the virtual address space inside the enclave and then calling the Keystone Security Monitor (SM) to initialize the enclave. The SM sets the permission bits on the PMP registers so that the memory region is not accessible to the untrusted OS. The SM then hashes all the pages in the enclave address space (including page table permissions) to compute the enclave identity. The SM then invokes the applet entry point, which is the decryption stub. The stub first requests an enclave report from the SM. The stub then delivers the report to the `TAPDanceManager` over an attested TLS connection (Step 3.1.b), which verifies the report matches the expected code-identity of the applet (Step 3.1.c). Recall that the `TAPDanceManager` obtained the expected applet code-identity over an attested TLS connection with the trusted client when the user created the applet. This solves our challenge of verifying the enclave report without needing the user to be online at the moment the applet runs.

If the enclave report is correct, the `TAPDanceManager` retrieves the keys for this applet and returns them to the applet enclave over an attested TLS connection (Step 3.1.d). These keys are k^u_{App}, k^u_{TS}, k^u_{AS}. Using k^u_{App}, the decryption stub will decrypt applet code, while the other two keys will allow that code to decrypt trigger data and later on, encrypt action data.

Cold vs. Warm Start. The process we have described is a cold applet enclave start. As our applet enclaves are small (about 2.2 MB of RAM), the untrusted TAP can keep them in memory so that when an event comes in, steps 3.1.a – e are skipped (shaded bar in Figure 3).

2) Trigger Data Retrieval: While the applet is launching, in parallel, the untrusted TAP obtains a new random nonce from the Security Monitor (SM; Step 3.2.a) and then sends out a request for data (Step 3.2.b) using the trigger OAuth token, nonce and trigger identity blob. The trigger service saves that nonce (Step 3.2.c) and return encrypted trigger data (Step 3.2.d) under key k^u_{TS}. The encrypted response also includes a timestamp and the nonce that was just received.

As discussed in Section II, trigger services run a shim layer to make themselves compatible with the TAP. Concretely, the shim stores an event queue containing the latest N trigger data events. We modify this event queue to include a slot for the last-seen nonce from the TAP for each event. If the slot is empty for a particular event (e.g., when new trigger data has been pushed in to the queue), the trigger shim stores the nonce from a data request for that event. If the slot is occupied for a particular event, the trigger shim ignores the nonce present in request. The current contents of the queue along with the respective nonce values are encrypted and returned to the TAP. Trigger services drop events and their nonce values from the end of the queue when it overflows.

3) Applet Execution: When the untrusted TAP receives encrypted trigger data, it places the data in a memory buffer accessible to the applet enclave and then requests the SM to jump to the enclave. The enclave decrypts using k^u_{TS}. If the decryption fails, the enclave returns an error. If decryption succeeds, it the applet extracts the nonce and asks the SM to verify it (Step 4). If nonce verification is successful, the applet verifies that the timestamp is within a time-to-live range using secure time from the monitor.

Once it has verified that the trigger data is fresh, the applet executes its rule on the trigger data to produce action data encrypted using k^u_{AS}. It includes an action-nonce inside this data, generated with secure randomness from the monitor, and another timestamp (Step 5). Finally, the applet enclave returns this encrypted action data to the untrusted TAP (Step 6) which forwards it to the action service (Step 7).

The action service will attempt to decrypt the action data using k^u_{AS}. If decryption is successful, it verifies that the timestamp in the message is within a time-to-live range and the action-nonce is not in its history of nonces. If true, it finally runs the action (Step 8). The action service automatically updates the history of last-seen nonces depending on its own timekeeping.

VI. Security Analysis

Confidentiality of Trigger and Action Data. The trigger shim only sends encrypted trigger data (using k^u_{TS}). Therefore, the untrusted TAP cannot misuse the OAuth token to steal trigger data. However, it does learn that a specific trigger event has occurred, and depending on the semantics of the trigger, this can leak information (e.g., a 1-bit event such as the WiFi being turned off). Investigating whether such leaks can be prevented in the TAP paradigm is potentially future work. The untrusted TAP could use its action OAuth token to misuse the action service. However, that service only performs actions if the payload is encrypted under a valid key (k^u_{AS}). The attacker does not have access to these keys are they are accessible only in the following components: (1) user's trusted client; (2) trigger/action shims; (3) `TAPDanceManager`. The applet enclave will get access to these keys after successfully passing an attestation check. The `TAPDanceManager` returns keys to applets based on the code identity, and thus, an attacker's applet would never get access to keys of another user/applet.

The service provider gives out OAuth tokens to the TAP and internally maintains a map between unique token strings and the identity of the TAP service. When anyone in possession of those unique tokens makes a request, the service provider looks up the map to determine the real identity. If an attacker uses stolen TAP OAuth tokens, the service provider will always know that the token was given out for TAP purposes and will subject the request to encryption checks.

Integrity of Applet Execution. As discussed in Section IV, relative to current TAP design, TAPDance provides: (1) isolation between enclaves (2) fault isolation of an untrusted enclave from the OS and (3) inverse sandboxing protecting the enclave from an untrusted OS, all using RISC-V physical

memory protection primitives. This guarantees applet code integrity.

The untrusted TAP can manipulate trigger API call parameters. TAPDance requires that the trigger shim also encrypt the API call parameters it received in its response. This way, the applet enclave will decrypt the response and verify that the API call parameters are the expected ones, and not something else. TAPDance adds these safety checks to applet code automatically as part of the compilation process.

The `TAPDanceManager` provides the keys to an enclave only after verifying its code-identity, obtained by applet from the SM and transmitted over an attested TLS connection. In Keystone, the enclave can only request its own report so an untrusted TAP cannot impersonate an enclave. Bugs could be present in the OCall wrappers that, if exploited, could allow the attacker to manipulate an enclave. We used defensive coding techniques to minimize this risk, such as checking bounds and return values.

Single Execution Per Trigger Event and Freshness of Data. The untrusted TAP can attempt to replay the encrypted data in TAPDance with the goal of (1) Executing the same applet multiple times with the same trigger event data; (2) Executing an applet with stale data; and (3) Executing actions multiple times even though there was only a single trigger event.

The untrusted TAP can request multiple copies of the same event data from the trigger service. However, the trigger shim associates the first nonce it sees from the TAP with the newly-available trigger data. If the TAP creates its own nonce or reuses a valid nonce from the SM, the SM will fail verification. Unless new event data is available, additional data requests from the TAP with new nonces (even from the SM) result in the shim returning event data with the original nonce, which prevents replay of existing events because the SM ensures that a particular nonce is verified only once by a specific enclave.

The untrusted TAP could delay the delivery of trigger data to the enclave. To counter this, the encrypted trigger data includes a timestamp which is checked by the applet enclave to fall within a freshness time window using the secure time provided by the SM. If the check fails, then the applet enclave ignores the trigger event without processing it and requests the untrusted OS for fresh trigger data.

TAPDance Interfaces to Untrusted OS. Table III shows all the interfaces that are available to enclave applications in TAPDance. The **`sendBufferFD`** and **`recvBufferFD`** OCalls are used by WolfSSL to send and receive data over a socket connection. We use defensive coding practices on the runtime-side to ensure that the OS is not returning nonsensical values for these calls. The SM calls provide secure randomness, secure time and nonce verification to the enclaves.

TEE Side Channels. The Keystone [48] framework considers three types of side channels: (i) Controlled channel (ii) Timing based and (iii) Cache based. Controlled channel attacks induce page faults in the enclave to learn about page access patterns. Keystone enclaves do not share page table state with the untrusted OS, thus eliminating this threat by design. To mitigate timing attacks against the cryptographic code inside an applet enclave, we enable timing resistance in WolfSSL during compilation. This only leaves timing attacks against applet code as a possibility. To mitigate cache attacks, Keystone recommends using cache partitioning [48] — we leave implementing this to future work.

```
var season = Meta.currentUserTime.month();
var sunrises: Array<number> =
[9, 8, 7, 7, 6, 5, 5, 6, 7, 8, 8, 9];
var sunsets: Array<number> =
[15, 16, 17, 19, 20, 21, 21, 20, 19, 18, 16, 15];
var hour = Meta.currentUserTime.hour();
if (hour >= sunrises[season] &&
     hour <= sunsets[season]) {
  Hue.turnOnAllHue.skip();
}
```

Figure 4: Example Benchmarking Applet

VII. Performance Evaluation

We measure the performance of TAPDance across four dimensions: (1) Number of applets supported (2) Reduction in software TCB; (3) Latency experienced by end-users while programming applets; (4) The Execution Protocol in terms of the end-to-end Applet Execution latency and throughput on the TAPDance server; (5) The Warm Execution Latency of an enclave after applet code and the required encrypted trigger data has been loaded into it.

Testbed. TAPDance relies on the RISC-V Keystone enclave mechanism, so we run TAPDance on the StarFive VisionFive single board computer. This machine is similar in power to a Raspberry Pi, so it is much slower than a server-class machine typically hosting internet services. There are recent plans for server-class RISC-V hardware, but nothing is currently available for purchase [49]. The VisionFive consists of 2 RISC-V U74 cores running at 1.5 GHz with 8 GB of RAM. Each U74 core from SiFive has support for 8 PMP regions [12]. The VisionFive runs Linux v5.19. To get Keystone running on the VisionFive, we ported the Security Monitor (SM) and compiled the Keystone Linux driver. We run the `TAPDanceManager` and applet enclaves on this VisionFive board. We use CloudLab to simulate trigger and action services [7]. The CloudLab machines have a 32 core Intel Xeon E5-2630 processor running at 2.4 GHz with 128 GB RAM. The VisionFive board is connected to the CloudLab machines through a Gigabit internet connection. For comparison, we run a baseline TAP system on the VisionFive using a Node.js v14.8.0 server that runs applets written in TypeScript.

Implementation Notes. We implement the TypeScript-to-assembly compiler on top of the StaticScript project on GitHub [13]. The compiler converts TypeScript to LLVM-IR which is then processed using the LLVM *llc* tool to generate a library containing the applet binary code. This applet library is linked with the enclave startup code to build the final enclave executable. We use the WolfSSL TLS library inside the applet enclave and the TAPDance Manager enclave [1], and configure it to use secure randomness and secure time from the SM. We configure WolfSSL to send TLS data using Ocalls to the untrusted OS. TAPDance uses the RapidJSON library [5] for parsing JSON data in the applet enclave. The trigger

OCalls Accessible to Enclaves	
`int initConnection(char *hostname, int port)`	Initiates a TCP connection to hostname:port returning the socket descriptor
`int initServerConnection(char *hostname, int port)`	Sets up a bound, listening socket returning the socket descriptor
`int sendBufferFD(int fd, char *buffer, uint64_t size)`	Write at most size bytes from the buffer to file descriptor fd and return the number of bytes written
`int recvBufferFD(int fd, char *buffer, uint64_t size)`	Read at most size bytes from the file descriptor fd to the buffer and return the number of bytes read
`char *getTriggerData(char *triggerParams, uint32_t *size)`	Gets Encrypted Trigger Data using triggerParams and returns the size of the trigger data
`char *sendActionData(char *actionParams, struct e_data *data)`	Sends encrypted and integrity protected action data
`int termConnection(int fd)`	Terminate the network connection referred to by fd
SM calls Accessible to Enclaves	
`int verifyNonce(uint64_t nonce)`	Verify the nonce is generated by the SM and has not been verified by another enclave with the same identity
`uint64_t getUnixTime()`	Returns the UNIX epoch time in seconds
`int getRandomBytes(char *buffer, size_t size)`	Fill a buffer with hardware provided randomness

Table III: Interfaces in TAPDance.

API and action API are thin wrappers over RapidJSON for accessing and modifying the various fields of the trigger data and action data, respectively. The functionality needed from the Moment.js library and the Date modules are implemented in C++ as a part of the applet API shim library. The host process that backs an enclave in Keystone forms the untrusted TAP in our setup. The host process uses the Crow HTTP server library to spawn a double threaded HTTP server on port 80 that listens for trigger event notifications from the trigger service [9].

Dataset. We obtained a dataset of real user applets from the authors of minTAP [32]. Each applet has an attached JSON schema that describes the input format the applet expects. This is the largest known dataset of TAP applet code. An example applet in the evaluation is shown in Figure 4 and in the Appendix(Figure 6). To run our performance experiments, we randomly chose 10 applets from the dataset.

Functionality Evaluation. Our TypeScript compiler successfully compiled 642 out of 682 applets from the minTAP dataset. The 39 applets that did not compile make use of user-defined objects, parameterized strings, union types and anonymous functions. Out of the 39, 37 can readily be re-written using our subset of TypeScript without loss in functionality. The remaining 2 applets make extensive use of user-defined objects and classes as well as indexing into the trigger data object that we do not support.

To verify correctness of the compiled code, we compiled and ran the 10 applets that were chosen for the performance evaluation. For each applet we generate a fixed input by adhering to the JSON schema and the typical response generated by the particular trigger service for the event. We verified that their outputs with TAPDance matched that from the NodeJS interpreter.

A. *TCB Size of Enclave Components*

We compare the TCB size of all the components in an enclave in TAPDance against a TAP system where TypeScript applets are using the Node.js interpreter inside a Keystone enclave (i.e., the "big enclave" approach is our baseline in this specific case). Table IV shows the LoC for TAPDance. WolfSSL is 183 KLoC for the default compilation, however, this is an upper bound as many of the cipher suites can be omitted from compilation using appropriate compilation flags. Similarly, RapidJSON is a feature rich JSON parsing library that is 18 KLoC in size.

In contrast to the TAPDance TCB, the baseline TAP contains a Node.js frontend (109 KLoC) and a V8 backend (935 KLoC) of C++ code. Additionally, the baseline approach delegates all system calls to the untrusted TAP operating system. Overall, TAPDance reduces the TCB compared to baseline by 5.2x and further optimizations are possible by trimming the TLS and JSON libraries.

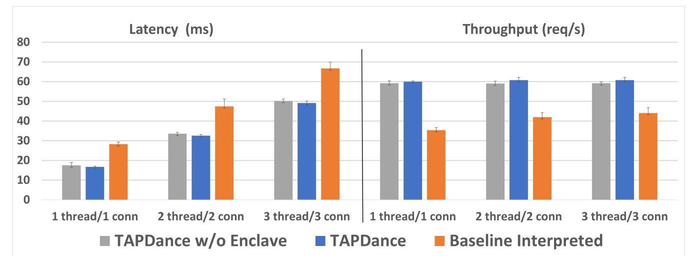

Figure 5: Latency and Throughput Comparison of TAPDance with the Baseline TAP that does not use enclaves. TAPDance performs better than the interpreted baseline, and identical to a version without enclaves.

Component	Lines of Code
RapidJSON	18496
Applet API Shim	378
Applet Enclave Init	1277
Eyrie Runtime	3365
WolfSSL	183000
Keystone LibEdge	384
Keystone LibApp	633
Ed25519 Verify	3119
Keystone SM	3040
Total	213692
Node.js Frontend	109 KLoC
V8 Backend	935 KLoC

Table IV: Code Sizes of Applet Enclave Components and in TAPDance and the Node.js Interpreter

System	Average Applet Execution Time
TAPDance	0.58 ms (SD = 0.19 ms)
TAPDance w/o enclave	0.57 ms (SD = 0.10 ms)
Interpreted Baseline	0.62 ms (SD = 0.18 ms)
System	**Resident Set Size**
TAPDance Applet Enclave	2.1 MB
TAPDance Manager Enclave	3.1 MB
Baseline	39.3 MB

Table V: Performance Metrics of TAPDance.

B. Performance

1) Trusted Client Latency: We measure the time it takes for the compiler to compile a TypeScript applet to a binary and also the time taken by the trusted client application, a C program, to register the applet with the TAPDance server and register the user if this is their first applet registered. Both the compiler and the trusted client run on a laptop having an 8 core Intel Core i7-7700HQ processor running at 2.80 GHz with 16 GB RAM and connected to the same Gigabit local network as the VisionFive board. The client app connects to `TAPDanceManager` over an attested TLS connection. All numbers are averaged over 5 trials for 10 applets chosen randomly from the dataset of real user-created applets. To register a user, the latency is 11 ms (SD = 0.34 ms) and to register the compiled applet, the latency is 11 ms (SD = 0.041 ms). The average time to compile the applet is 398 ms (SD = 25 ms).

When an applet is not in memory when trigger data arrives, the cold-start cost of memory allocation, communication with the `TAPDanceManager` and enclave creation takes 200ms, and measuring the enclave in the SM takes 3s on our prototype board. This reduces to 1ms with hardware hashing support present in modern Intel processors.

2) Applet Execution Time: We measured the *warm execution latency*, when an applet is already loaded into an enclave, as the time it takes to run the applet enclave after the untrusted TAP fetches the encrypted trigger data. This time, shown in Table V, averages 0.58 ms (SD = 0.19 ms) for the 10 test applets. In comparison, the time taken to run an applet in the baseline Node.js server averages 0.62 ms (SD = 0.18 ms). TAPDance is faster than the baseline because we compile applet code to assembly and, thus, remove interpreter overhead present in the baseline. The enclave overhead is very low; if we run the compiled applet without an enclave the average latency is nearly identical to TAPDance at 0.57 ms (SD = 0.10 ms). We attribute this to the fact that there are at most 3 enclave-to-host context switches per applet execution and each enclave switch takes 3 microseconds.

3) End-to-End Performance: To measure the execution protocol performance, i.e., throughput and latency of TAPDance, the baseline TAP server and running TAPDance without enclaves, we generate trigger events using the *wrk* [2] HTTP benchmarking tool. We vary the rate of trigger events by varying the number of threads and connections used for generating trigger events. We run each system for 10 seconds with varying loads. The *wrk* tool is run from a machine on CloudLab. We executed the TAPDance execution protocol to the point before sending the action data and send a notification back to the trigger event generator (*wrk*). We varied the event generation rates to find the highest degree of parallelism that both systems support before performance degrads. We average the results over the same 10 chosen applets for each system.

Figure 5 summarizes the latency and throughput for TAPDance and the interpreted baseline by varying the number of connections and threads. TAPDance has 32% lower latency than the baseline on average and has a 33% higher throughput than the baseline. As stated earlier, this difference is because TAPDance runs compiled applets whereas the baseline runs interpreted applets. This shows that secure execution is not necessarily higher overhead than non-secure execution.

4) Memory Usage: We measure the amount of memory that is needed to run the applet enclave and the `TAPDanceManager`. Enclaves have constant size and cannot be expanded or shrunk dynamically in TAPDance.

The applet enclave takes 2.1 MB (533–537 pages) and the `TAPDanceManager` enclave takes 3.2 MB (816 pages). In contrast, the Node.js interpreter has a resident set size of 39 MB. With a much lower memory footprint, TAPDance can keep many more applets warm in memory and avoid paying the cold start cost. For example, with 128 GB of memory a server could keep about 62,000 applets warm. With contiguous allocation, a single PMP register can protect warm applet enclaves from access by the untrusted OS.

VIII. Discussion and Limitations

Supporting Multiple Triggers, Actions and Queries. Currently, TAPDance supports a single trigger service and multiple action services for a particular applet. It is possible for applets to trigger on multiple data sources. Our design naturally extends to this scenario by creating separate trigger symmetric keys for each service and then storing those keys in the `TAPDanceManager`, just like we do for multiple actions. We would need to modify the applet enclave stub to wait for multiple trigger data requests to return before running the computation. We leave implementing this to future work.

Scalability of TAPDance. TAPDance currently supports running applet enclaves in any machine with Keystone enclave support, provided that the TAPDance Manager is loaded with the device public keys of all the machines involved. A central challenge in the replication of the TAPDance Manager is reaching consensus on a set of keys that are used for encrypting and integrity protecting data in untrusted storage. Sealing keys provide a way for a single enclave to derive a key, but these keys are tied to the processor package that the enclave is currently running on, and they cannot be re-derived by the same enclave code running in a different processor package in the data center. We design a protocol for TAPDance Manager replication by noting the separation between untrusted storage and the TAPDance Manager enclave and describe it in Appendix B.

Key Update Protocol. TAPDance relies on long-term keys for the trigger and action services and for encrypting the applet code. If the keys need to be replaced, the user will have to re-register the applet with the `TAPDanceManager` with different keys and then update the endpoint services with those new keys.

Re-ordering of Messages in the Action Service. TAPDance does not protect against reordering of messages to the action service as it uses nonces for replay detection and has no sequence numbers for ordering. We leave implementing message ordering guarantees as future work.

Publicly Influenced Triggers. Applets can consume publicly influenced trigger data. For example, an applet that triggers on an email is publicly influenced because anyone can send an email to a user with such an applet. An attacker could send a specially crafted email that exploits an applet with vulnerable code generated by our TypeScript compiler. Due to the mutual-distrust property of TAPDance, the other users' enclaves remain protected. However, the vulnerable applet can leak its trigger and action keys to the attacker. We acknowledge this limitation of TAPDance and observe that it is mitigated to some degree because the applet enclave runs a minimal trusted computing base. As future work, we anticipate verifying that the compiled code does not introduce exploitable vulnerabilities [47], [52].

Availability of server-grade RISC-V. We evaluate TAPDance on an under-powered RISC-V development board because no server-class chips are currently available. This does not affect the results because we report performance data relative to a baseline that also runs on the under-powered board. This also does not affect the technical contributions as we only rely on PMP registers, a standard element of the RISC-V instruction set architecture that will be available on all chips. Vendors have recently announced server-grade RISC-V chips [49].

IX. Related Work

Trigger-Action Platform Security. We contribute to a line of work on re-imagining the security properties of TAPs [19], [24], [32]–[34], [38], [63]. Table VI compares of TAPDance with previous work. TAPDance occupies the best trade-off among the design parameters of security, functionality and performance compared to existing approaches, when the attacker is the TAP environment itself. It offers confidentiality and integrity of real user applets with lower performance overhead than eTAP and Walnut, the closest related systems. The performance gain comes primarily from using TEEs, as opposed to techniques for computing on encrypted data. Furthermore, TAPDance supports a restricted TypeScript language for programming applets. We find that this language is sufficient to express 642/682 applets in the minTAP dataset and is amenable to static compilation. Whereas, eTAP/Walnut only supports applets that can efficiently be expressed as garbled circuits with fixed input sizes and unrolled loops. Executing garbled circuits requires several thousand symmetric key operations (depending on circuit size), making it much less efficient compared to native code. Finally, the action service in eTAP needs to maintain state about expected circuit IDs of all applets for all users as opposed to TAPDance that stores a nonce for only a freshness window. We note that the security of TAPDance is conditioned on a different set of primitives. eTAP/Walnut rely on cryptography (garbled circuits), whereas TAPDance relies on the correct operation of the physical memory isolation of RISC-V.

The primary goal of TAPDance is to provide the confidentiality of trigger and action data, as well as the integrity of action data computation. TAPDance offers stronger security than minTAP because plaintext data is never accessible to the untrusted TAP. Although minTAP releases *sanitized* trigger data to the untrusted TAP, it is still sensitive user data that is released. Additionally, minTAP does not provide integrity guarantees on applet execution. OTAP end-to-end encrypts trigger data, but it does not support computing on that data [34]. DTAP only supports action integrity without computation on trigger data and does not provide data confidentiality [38]. These systems have better performance than TAPDance, but have strictly lower security guarantees and support lesser TAP program functionality.

A common theme in prior work is the concept of decentralizing trust by introducing a trusted client device [32], [33], [38]. We borrow this idea because it helps minimize the TCB inside the enclave. The alternative solution would be to have an enclave host code for programming applets, which increases the complexity of enclave code.

An orthogonal line of work investigates the security and privacy properties of applet logic [35], [59], [63]. For example, applets can unintentionally leak private user data to a public source. We do not rely on applet logic for security guarantees; our goal is to protect an applet, independently of its semantics, from a malicious TAP environment and from other possibly malicious applets.

TAPs suffer from overprivilege primarily because OAuth makes it difficult to enforce the principle of least privilege. Protocol improvements like macaroons can help [26], but they provide a weaker security guarantee than what we are aiming for in this work. By contrast, TAPDance enforces that the trigger only transmits encrypted data and the action service only accepts encrypted payloads, a higher level of security because sensitive data is not accessible to the TAP in plaintext, except inside an attested enclave.

Enclave-based Systems. VC3 [56] demonstrates how Intel SGX enclaves can be used to secure MapReduce computations. Similar to TAPDance, it uses a model where a verifier attests all the worker nodes in the MapReduce computation. However, the verifier needs to be maintained by the user running the MapReduce job unlike TAPDance where the verification is delegated to the `TAPDanceManager`, letting a user be offline when a trigger event fires.

Clemmys [60] provides a framework for running serverless functions in SGX enclaves. The functions are run inside a SCONE-based environment [22], that supports unmodified POSIX binaries. Similar to TAPDance, Clemmys relies on an attestation service like the `TAPDanceManager`. Serverless functions are short-lived and are similar to the applets executed in Trigger-Action Platforms. However, TAP applets are pure computations and do not require the extensive system call support provided by SCONE. This allows TAPDance to keep a relatively low software trusted computing base involving a TLS library and a customized userspace runtime to service applet library functionality needs. In general, serverless frameworks that use TEEs have large runtimes relative to TAPDance because they are designed to support general computations instead of pure computations [20], [27], [54], [60]

Komodo [39] decouples the enclave memory management

Criterion	IFTTT	eTAP [33], Walnut [55]	MinTAP [32]	DTAP [38]	OTAP [34]	TAPDance (our work)
`Confidentiality of Trigger Data`	No	Yes	Minimized Access	No	Yes	Yes
`Integrity of Action Computation`	No	Yes	No	Partial	No	Yes
`Performance Overhead`	Baseline	High	Medium	Low	Low	Outperforms Baseline
`Support for Applets`	All	Restricted to Circuits	All	None	None	Restricted TypeScript
`Freshness Guarantees on Trigger`	No	Yes	No	Yes	No	Yes
`Action Replay Protection`	No	Yes	No	No	No	Yes

Table VI: Comparison of TAPDance with prior TAP security systems. TAPDance offers the best trade-off among all design criteria.

from the hardware, in contrast to SGX. The memory management is delegated to a formally verified software monitor that runs in the ARM secure mode. In Keystone, enclaves are in charge of managing their own memory using the runtime within the enclave. We did not use Komodo in TAPDance mainly because applets do not require dynamic memory management. The S-Mode runtime in TAPDance aids in handling in-enclave faults from within the enclave, without leaking information to the untrusted OS.

Ryoan [41] provides a way to define dependencies for a particular computation as a DAG between multiple third party services which are mutually untrusted. Unlike TAPDance, the attestation of each instance of the sandbox is done by the user. Ryoan does not attempt to reduce the TCB inside the enclave, unlike TAPDance that, uses the specific properties of TAP computing to reduce the TCB.

Panoply [58] is a framework that lets users run modified programs inside SGX enclaves with a minimal TCB. The user has to run an analysis that annotates the entry and exit points of enclave code, and only the shielding code for the POSIX APIs needed by the enclave code is included in the enclave. This TCB reduction approach is similar to TAPDance. However, TAPDance does not require the syscall, I/O and concurrency support provided by Panoply.

Graphene [30] and Haven [25] are library OSes whose goal is to run unmodified binaries inside an SGX enclave. They are orthogonal to the goals of TAPDance, in that they include as much functionality as required to run unmodified binaries inside the enclave, amounting to several million lines of code.

X. Conclusion

Trigger-action platforms are increasingly critical as an automation tool for coordinating multiple web-based services. Current TAP architectures are fundamentally insecure as they expose trigger and action data to untrustworthy TAP service code. Our key insight is that TAP applets are pure functions, and in this work we show how to leverage this property to build a secure scalable TAP. Our design uses the unique protection hardware on RISC-V and its Keystone security monitor, along with novel protocols, to provide data privacy and applet execution integrity. Our evaluation demonstrates that the TAPDance design offers substantial security benefits at almost no runtime performance loss.

Acknowledgements. We thank the anonymous reviewers, our anonymous shepherd, Andrei Sabelfeld and Yunang Chen for valuable feedback. This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and in part by NSF grants CNS 1763810, 2144376, 2312119 and a gift from Amazon.

References

[1] “WolfSSL Embedded TLS Library,” https://www.wolfssl.com/, 2012.

[2] “wrk HTTP Benchmarking Tool,” https://github.com/wg/wrk, 2012.

[3] “Xsa-108: Improper msr range used for x2apic emulation,” http://xenbits.xen.org/xsa/advisory-108.html, Oct. 2014.

[4] “Database of 191 million u.s. voters exposed on internet.” https://www.reuters.com/article/us-usa-voters-breach-idUSKBN0UB1E020151229, Oct. 2015.

[5] “RapidJSON: A fast JSON parser/generator for C++ with both SAX/DOM style API,” https://rapidjson.org/, 2016.

[6] “Uber employees ’spied on ex-partners, politicians and beyoncé’,” https://www.theguardian.com/technology/2016/dec/13/uber-employees-spying-ex-partners-politicians-beyonce, Oct. 2016.

[7] “CloudLab,” https://www.cloudlab.us/, 2017.

[8] “AssemblyScript: A TypeScript-like language for WebAssembly,” https://www.assemblyscript.org/, 2020.

[9] “Crow: A Fast and Easy to use microframework for the web,” https://crowcpp.org/master/, 2020.

[10] “IFTTT: If This Then That,” https://ifttt.com, 2020.

[11] “OP-TEE: Open Portable Trusted Execution Environment,” https://www.trustedfirmware.org/projects/op-tee/, 2020.

[12] “SiFive U74 Core,” https://www.starfivetech.com/uploads/u74_core_complex_manual_21G1.pdf, 2020.

[13] “StaticScript TypeScript Compiler,” https://github.com/ovr/StaticScript, 2020.

[14] “Cve-2021-44228 detail (log4shell),” https://nvd.nist.gov/vuln/detail/CVE-2021-44228, Dec. 2021.

[15] “Github breach 2022,” https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/, Apr. 2022.

[16] “IFTTT Documentation,” https://ifttt.com/docs/api_reference, 2022.

[17] “Circleci breach 2023,” https://circleci.com/blog/january-4-2023-security-alert/, Jan. 2023.

[18] “Report: Activision failed to tell employees of 2022 data breach,” https://www.gamedeveloper.com/culture/report-activision-blizzard-failed-to-tell-employees-of-2022-data-breach, Feb. 2023.

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

[20] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, “S-faas: Trustworthy and accountable function-as-a-service using intel sgx,” in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, ser. CCSW’19, 2019, p. 185–199.

[21] AMD, “Amd secure encrypted virtualization (sev),” 2017, accessed on April 18, 2023.

[22] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Nov. 2016, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

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

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

[25] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Oct. 2014, pp. 267–283. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann

[26] A. Birgisson, J. G. Politz, Úlfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud," in *Network and Distributed System Security Symposium*, 2014.

[27] S. Brenner and R. Kapitza, "Trust more, serverless," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 33–43. [Online]. Available: https://doi.org/10.1145/3319647.3325825

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

[29] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi, "Type inference for static compilation of javascript," *SIGPLAN Not.*, vol. 51, no. 10, p. 410–429, oct 2016. [Online]. Available: https://doi.org/10.1145/3022671.2984017

[30] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Jul. 2017, pp. 645–658. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[31] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 253–264. [Online]. Available: https://doi.org/10.1145/2451116.2451145

[32] Y. Chen, M. Alhanahnah, A. Sabelfeld, R. Chatterjee, and E. Fernandes, "Practical data access minimization in Trigger-Action platforms," in *31st USENIX Security Symposium (USENIX Security 22)*, Aug. 2022, pp. 2929–2945. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-yunang-practical

[33] Y. Chen, A. R. Chowdhury, R. Wang, A. Sabelfeld, R. Chatterjee, and E. Fernandes, "Data privacy in trigger-action systems," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 501–518.

[34] Y.-H. Chiang, H.-C. Hsiao, C.-M. Yu, and T. H.-J. Kim, "On the privacy risks of compromised trigger-action platforms," in *Computer Security – ESORICS 2020*, L. Chen, N. Li, K. Liang, and S. Schneider, Eds., 2020.

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

[36] djblend777, "Private links and photos from https://locker.ifttt.com - how to clear history?" https://www.reddit.com/r/ifttt/comments/ao3sfr/private_links_and_photos_from_httpslockerifttcom/, 2019.

[37] S. Duan, H. Meling, S. Peisert, and H. Zhang, "Bchain: Byzantine replication with high throughput and embedded reconfiguration," in *International Conference on Principles of Distributed Systems*, 2014.

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

[39] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, 2017, p. 287–305. [Online]. Available: https://doi.org/10.1145/3132747.3132782

[40] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, p. 1–14. [Online]. Available: https://doi.org/10.1145/2670979.2670986

[41] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, dec 2018. [Online]. Available: https://doi.org/10.1145/3231594

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

[43] IFTTT, "Important update about the Gmail service," https://help.ifttt.com/hc/en-us/articles/360020249393-Important-update-about-the-Gmail-service, 2019.

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

[45] ——, "IFTTT: Service API requirements," https://platform.ifttt.com/docs/api_reference, 2020.

[46] Intel, "Intel software guard extensions (intel sgx)," 2015, accessed on April 18, 2023.

[47] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Доверя́й, но проверя́й: SFI safety for native-compiled Wasm," in *NDSS*. Internet Society, 2021.

[48] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, 2020.

[49] S. Leibson, "SiFive Unveils 64-Bit RISC-V Server Core," https://www.eetimes.com/sifive-unveils-64-bit-risc-v-server-core/, 2021.

[50] J. A. Martin and M. Finnegan, "What is IFTTT? How to use If This, Then That services," Computerworld. https://www.computerworld.com/article/3239304/what-is-ifttt-how-to-use-if-this-then-that-services.html, 2019.

[51] "Microsoft Power Automate," https://flow.microsoft.com/, 2020.

[52] M. Patrignani, A. Ahmed, and D. Clarke, "Formal approaches to secure compilation: A survey of fully abstract compilation and related work," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[53] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "Sgx-lkl: Securing the host os interface for trusted execution," 2020.

[54] W. Qiang, Z. Dong, and H. Jin, "Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave," in *Security and Privacy in Communication Networks*, R. Beyah, B. Chang, Y. Li, and S. Zhu, Eds. Cham: Springer International Publishing, 2018, pp. 451–470.

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

[56] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 38–54.

[57] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, mar 2020. [Online]. Available: https://doi.org/10.1145%2F3373376.3378469

[58] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with SGX enclaves," in *24th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, Mar. 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/panoply-low-tcb-linux-applications-sgx-enclaves/

[59] 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*, 2017, pp. 1501–1510.

[60] B. Trach, O. Oleksenko, F. Gregor, P. Bhatotia, and C. Fetzer, "Clemmys: Towards secure remote execution in faas," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR '19, 2019, p. 44–54. [Online]. Available: https://doi.org/10.1145/3319647.3325835

[61] H. Tschofenig, T. Fossati, P. Howard, I. Mihalcea, and Y. Deshpande, "Using Attestation in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," Internet Engineering Task Force, Internet-Draft draft-fossati-tls-attestation-02, Oct. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-fossati-tls-attestation/02/

[62] R. Van Renesse and F. Schneider, "Chain replication for supporting high throughput and availability." 01 2004, pp. 91–104.

[63] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, 2019. [Online]. Available: https://doi.org/10.1145/3319535.3345662

[64] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 640–656.

[65] "Zapier," https://zapier.com, 2020.

[66] Zapier, "Zapier Platform CLI Docs," https://platform.zapier.com/cli_docs/docs, 2020.

APPENDIX

A. User Registration and Authentication to TAPDanceManager

The Algorithms 1 and 2 describe the process of a user registering and then authenticating to the `TAPDanceManager`.

Algorithm 1 Keystore.UserRegistration()

Input: $username_{user}, password_{user}$
Trusted Inputs: $username_{user}, password_{user}$
Connection Channel: Attested TLS to Keystore
1: $y \leftarrow UntrustedDataStore.GetUser(username_{user})$
2: **if** y exists **then**
3: **return** "User Exists"
4: **end if**
5: $id_{user} \leftarrow GetUID(username_{user}, password_{user})$
6: $k_{seal} \leftarrow SM.DeriveSealingKey(password_{user})$
7: $IV \leftarrow SM.GetRandomBytes(16)$
8: $(C_{user}, T_{user}) \leftarrow AES_Encrypt(k_{seal}, IV, nil, \{id_{user}, username_{user}\})$
9: $UntrustedDataStore.Store(< username_{user}, (C_{user}, T_{user}, IV) >)$
10: **return** id_{user}

B. Replication of TAPDanceManager

The ability to dynamically provision and run multiple instances of `TAPDanceManager` is crucial for scaling TAPDance with an increasing number of deployed applets. Recall that each applet enclave needs to contact the `TAPDanceManager` at startup for obtaining the keys for successful operation. Further, the `TAPDanceManager` may fail and we also want high availability of TAPDance.

A `TAPDanceManager` instance can be visualized as the `TAPDanceManager` enclave coupled with an untrusted storage node. As the `TAPDanceManager` enclave is attested

Algorithm 2 Keystore.UserAuthentication()

Input: $username_{user}, password_{user}$
Trusted Inputs: $username_{user}, password_{user}$
Connection Channel: Attested TLS to Keystore
1: $y \leftarrow UntrustedDataStore.GetUser(username_{user})$
2: **if** y does not exist **then**
3: **return** "User Does Not Exist"
4: **end if**
5: $(C_{user}, T_{user}, IV) \leftarrow y$
6: $k_{seal} \leftarrow SM.DeriveSealingKey(password_{user})$
7: $dec \leftarrow AES_Decrypt(k_{seal}, IV, C_{user}, T_{user}, nil)$
8: **if** dec is not valid **then**
9: **return** ("Unsuccessful", nil)
10: **end if**
11: $(id_{user}, username_{user}) \leftarrow dec$
12: **return** ("Successful", id_{user})

and the code is available publicly, it is assumed to not exhibit Byzantine behavior. However, the untrusted storage node does exhibit Byzantine behavior.

We use chain replication [62] to achieve replication of a `TAPDanceManager` instance. Each `TAPDanceManager` instance is part of a chain. Failures of chain elements are detected by a service provided by the untrusted cloud infrastructure. This failure information is used by the chain elements to reconfigure themselves. The client contacts the failure detection service to locate the head of the chain. The head of the chain processes update requests (applet registration) and passes the updates to its successor. Once the tail processes the propagated updates, a reply is sent to the client. Queries for data are sent to the tail. Each instance connects to its successor using attested TLS. Prior to operation, the head of the chain generates a shared key $k_{storage}$ that is propagated to all the instances in the chain. $k_{storage}$ is used to encrypt and integrity protect the (i) user credentials and (ii) the applet credentials before storing it in the attached storage node.

BChain [37] has explored the setting where nodes in chain replication exhibit Byzantine behavior. In BChain, a certain fraction of the nodes are assumed to be faulty and uses a timeout based faulty node detection mechanism to move faulty nodes out of the chain. In TAPDance, the key difference from BChain is that the `TAPDanceManager` enclaves that form the logical elements of the chain do not exhibit Byzantine faults. However, the `TAPDanceManager` enclave maybe be induced to produce Byzantine faults either if (i) the attached storage node produces Byzantine faults or (ii) if the failure detection service produces Byzantine faults and lies to the replicas about the chain configuration.

Our key observation is that even in the presence of Byzantine faults, data corruption leads to a denial of service condition as opposed to an attack on TAPDance.

If the storage node corrupts the stored ciphertext for an applet, then it does not decrypt correctly in the `TAPDanceManager` enclave. The only case where the storage node can return data that decrypts correctly is if it either returns the expected ciphertext or if it returns ciphertext that was generated using $k_{storage}$. The returned value could be (i) the ciphertext corresponding to a different applet, in which

```
var s_length = parseInt(AndroidPhone
            .placeAPhoneCall.CallLength);
var endTime = moment(moment(AndroidPhone
    .placeAPhoneCall.OccurredAt,
    'MMMM dd, YYYY at hh:mmA')
    .add(moment(AndroidPhone
    .placeAPhoneCall.CallLength, 'seconds')),
    'MMMM dd, YYYY at hh:mmA').toString();
var min = moment(moment(AndroidPhone
    .placeAPhoneCall.OccurredAt,
    'MMMM dd, YYYY at hh:mmA')
    .add(1,'minutes'),
    'MMMM dd, YYYY at hh:mmA').toString();

if (s_length > 120) {
  GoogleCalendar.addDetailedEvent
  .setEndTime(endTime);
} else {
  GoogleCalendar.addDetailedEvent
  .setEndTime(min);
}
```

Figure 6: Benchmarking Applet

case the hashes of the applet enclave do not match with the decrypted value in the applet information or (ii) the ciphertext corresponding to a stale version of the applet description, meaning that atleast one of k^u_{App}, k^u_{TS} or k^u_{AS} is different from the present version. If k^u_{App} is different, then the applet enclave will not decrypt correctly. Recall that a single set of trigger and action keys are in use at a time for a particular user. Hence, if either of k^u_{TS} or k^u_{AS} is different it would lead to an inability to decrypt the trigger data blob by the applet enclave or the inability to decrypt the encrypted action blob by the action service.

Similarly, the storage node could try and replay a stale ciphertext encrypting the credentials of a different user or the previously used (but changed) credentials of the user, during authentication. In the first case, the user will not be authenticated, and it results in a denial of service condition. In the second case, if the untrusted TAP has knowledge of the previous password, then it can authenticate itself as the user. Impersonating as the user, the untrusted TAP can only delete applets deployed by the user. Deploying a new applet needs knowledge of the trigger and action keys for the particular applet, and the untrusted TAP is assumed to not have knowledge of the same.

Lastly, the failure detection service can cause different instances to diverge in what they store in the untrusted store by lying about the state of the chain to the instances. However, this case results in exactly the same situation as if the untrusted storage node of the tail instance is misbehaving as described above.