Leveraging Execution Models for Designing Secure Services

by

Deepak Sirone Jegan

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: 07/30/2025

The dissertation is approved by the following members of the Final Oral Committee:

Michael M. Swift, Professor, Computer Sciences
Earlence Fernandes, Assistant Professor, CSE, UC San Diego
Somesh Jha, Professor, Computer Sciences
Ethan Cecchetti, Assistant Professor, Computer Sciences
Kassem Fawaz, Associate Professor, ECE

© Copyright by Deepak Sirone Jegan 2025
All Rights Reserved

Dedicated to everyone who believed in me over the years

Acknowledgments

Graduate school has been a great learning experience, and there are several individuals whom I wish to thank for continually supporting me over the past six years. Even before that, there were several people who built me up to succeed in a world-class graduate program.

First and foremost, I wish to thank my advisor, Prof. Michael Swift, for taking me on as an advisee and mentoring me to become a competent researcher. Mike has the rare combination of being really knowledgeable while being patient, kind, compassionate, and a terrific human being, always looking out for his students. Over the 100s of meetings that we had, Mike taught me to express the core of any complex idea as succinctly as possible without losing out on rigor. He also taught me to think broadly and look for solutions in unlikely areas with confidence. His patience in dealing with my bad writing skills has been truly legendary. Through the way he lives his own life, and his professionalism, he has indirectly taught me to live better and aim farther. Thank you, Mike, for investing so much time and effort into my growth and for teaching me how to think. It has been truly wonderful and an honor working with you.

Secondly, I would like to thank my thesis committee for their valuable guidance throughout my PhD. Prof. Earlence Fernandes has always been a well-wisher and helped me greatly in the TAPDance project. Prof. Somesh Jha helped me in the Kalium project by suggesting changes to the reasoning that we were using. Prof. Rishab Goyal taught me graduate cryptography and helped me in the Bellerophon project by recommending we use HIBE, which became a crucial design point. Prof. Kassem Fawaz provided me with valuable career guidance. Over the course of my PhD, I was fortunate to interact and learn from several great researchers: Prof. Ethan Cecchetti, Prof. Patrick McDaniel, Dr. Ryan Beckett, Prof. Tom Reps, and Prof. Rahul Chatterjee. Prof. Ethan Cecchetti helped me a lot in writing the Bellerophon

paper and in its subsequent modifications. Prof. Patrick McDaniel helped me by giving me directions for future research and also career advice. Dr. Ryan Beckett was my mentor during my internship at Microsoft Research, and he gave me the wonderful opportunity to prototype a new network verification product that is currently deployed in Microsoft Azure. Prof. Tom Reps taught me advanced compilers, which helped me in both my internship as well as in the TAPDance project. Prof. Rahul Chatterjee encouraged me to pursue systems security during the early days of my PhD, which turned out to be the right decision.

I was fortunate to have worked with a set of brilliant researchers in Mike's SCAIL research group, including Mark Mansi, Anjali, Sujay Yadalam, Bijan Tabatabai, Ashwin Poduval, Hayden Coffey, and Yusen Liu. Their feedback on each of my projects and presentations ensured that I was always on the right track. I am also fortunate to have met and interacted with several students from adjacent research groups: Chloe Alverti, Konstantinos Kanellis, Jing Liu, Anthony Rebello, Guanzhou Hu, Vinay Banakar, Sambhav Satija, Naman Gupta, Chenhao Ye, Shawn Zhong, Quinn Burke, Rachel King, Aishwarya Ganesan, and Ramnatthan Alagappan. Attending regular reading groups with these people helped me feel at home and also kept me informed on the various happenings in other research areas.

After arriving at UW-Madison, I was extremely fortunate to have met a set of friends who guided me and kept me happy and motivated throughout this journey. Roshan Xavier and Sreyas Mohan took me on a 40-mile bike ride within a week of arriving at Madison, even when I had never ridden a bike for more than 5 miles. This made me very confident in biking long distances throughout my time at Madison, a skill that I would never have acquired on my own. Eventually, I met more friends from Roshan's circle: Kaivalya Molugu and Debaditya Chatterjee. Together, they kept me sane for the first four years at Madison, through hikes, doing the

MOVE4BGC 50-mile bike ride twice, pickleball, volleyball at Brittingham Park, camping at Appleton, and cutting birthday cakes at our "secret spot" near Raymer's Cove.

The pandemic presented additional opportunities to connect with old high school friends and their friends: Nived Shibu, Atul Narayanan, Akhil Anil, Allen E. Baiju, Hafs V. Siraj, and Vivek Krishnan. Together, we formed the "Pineapple Farmers", a gaming group that specializes mostly in Counter Strike but occasionally dabbles in other games like Halo. With Atul being in Italy and everyone else (except me) in India, I was perpetually stuck with a 250+ ms ping, which made things interesting. Thank you for the many hours of fun, chaos, and lively conversation that were very energizing for me during the pandemic.

Special thanks to my mentors at the Badger Ballroom Dance Team: Colin Richter, Taylor Treadway, Lorenz Vargas, and Ross Urven. Although I wasn't very good at dancing or had never participated in a dance competition before, I was able to build the confidence to participate in the beginner tier at the Purdue Ballroom Classic 2021. Together with Savannah Olson, we reached the semi-finals of foxtrot, something that we never expected. During the year I spent regularly attending classes at BBDT, I was introduced to both Latin and Standard dances, and I am keen on continuing to learn after I graduate.

I am indebted to Varun Sundar, Padideh Hassanpour, Anthony Rebello, Siddharth Suresh, Sujay Yadalam, Naman Gupta, Sambhav Satija, Vinay Banakar, Gowtham Ramesh, and Karan Grover for all the dinners while here at Madison. Special thanks to Varun and Padideh for all the pickleball games, the sailing, bike rides, meetups at various restaurants, and all the help and kindness that you have given me during the past two years.

Prior to joining UW-Madison, I was fortunate to be mentored by excellent teachers. At IIT Kanpur, I was fortunate to be mentored by Prof. Pramod Subramanyan and Prof. Sandeep Shukla. Prof. Pramod's infectious

energy and encouragement led me to apply for a PhD, including letting me do research on formal methods, set questions for the Computer Security course Capture the Flag (CTF) competitions, and reading through my Statement of Purpose for university applications. Prof. Sandeep taught me the Computer Systems Security course which was instrumental in growing my interest in security research. At NIT Calicut, after a rather uneventful first year, I was delighted to be introduced to Computer Science by Prof. Vineeth Paleri, Prof. Muralikrishnan K., Prof. Priya Chandran, Prof. Vinod Pathari, and Prof. Sudeep K.S. Their combined strength in teaching taught me to approach theoretical and systems concepts. I am forever indebted to them for putting me on the path to pursue research.

Finally, my entire PhD journey would not have been possible without the support of Amma, Appa, and my sister Roshni. Thank you for bearing with me and always being there for me. I could not have achieved anything without you.

Contents

List of Tables x

List of Figures xi

Abstract xiv

1 Introduction 1

1.1 Execution Models 3

1.2 Security Challenges and Opportunities in Existing Cloud Execution Models 4

1.3 Contributions 8

1.3.1 Defining and Enforcing Control Flow Integrity for Serverless Applications 8

1.3.2 Architecting Trigger Action Platforms using Trusted Execution Environments for Security, Performance and Functionality 11

1.3.3 Non-Interactive Remote Attestation 13

1.4 Thesis Organization 15

2 Control Flow Integrity for Serverless Applications 16

2.1 Introduction 16

2.2 Background 19

2.2.1 Serverless data confidentiality and integrity 19

2.2.2 Serverless control flow 20

2.2.3 Information flow control in serverless 21

2.3 Motivation and Challenges 23

2.3.1 Attacks against serverless applications 23

2.3.2 Common design patterns of serverless applications 24

2.3.3 Challenges 26

2.3.4 *Limitations of existing approaches* 27

2.4 *Threat Model* 30

2.5 *Design of Kalium* 32

2.5.1 *Control flow tracking* 32

2.5.2 *Secure runtime* 38

2.5.3 *Controller* 41

2.5.4 *Extensions to Kalium* 41

2.6 *Implementation* 43

2.7 *Evaluation* 45

2.7.1 *Workloads* 45

2.7.2 *Efficiency of flow tracking* 47

2.7.3 *Security analysis* 50

2.7.4 *Performance overhead of Kalium* 52

2.8 *Limitations of Kalium* 57

2.9 *Related Work* 59

2.10 *Conclusion* 60

3 Architecting Trigger Action Platforms for Security, Performance and Functionality 61

3.1 *Introduction* 61

3.2 *Background* 65

3.3 *Design Considerations* 68

3.3.1 *Threat Model* 68

3.3.2 *Security Goals* 69

3.3.3 *Functionality Goals* 70

3.3.4 *Alternative Approaches* 70

3.4 *TAPDance Design* 71

3.4.1 *Challenges and Solutions* 73

3.4.2 *TAPDance Components* 75

3.4.3 *User Registration and Authentication to TAPDanceManager* 81

- *3.4.4 Replication of TAPDanceManager* 81
- *3.5 Creating and Running Applets with TAPDance* 86
  - *3.5.1 User Bootstrapping* 88
  - *3.5.2 Creating an Applet* 88
  - *3.5.3 Running an Applet* 89
- *3.6 Security Analysis* 94
- *3.7 Performance Evaluation* 97
  - *3.7.1 TCB Size of Enclave Components* 99
  - *3.7.2 Performance* 100
- *3.8 Discussion and Limitations* 104
- *3.9 Related Work* 106
- *3.10 Conclusion* 110

4 Non-Interactive and Verifier-free Remote Attestation 111
- *4.1 Introduction* 111
- *4.2 Background* 116
  - *4.2.1 TEEs and Remote Attestation* 116
  - *4.2.2 Hierarchical Identity-Based Encryption* 118
- *4.3 Design Considerations* 120
  - *4.3.1 Adversarial Model and Compromise Recovery* . . . 120
  - *4.3.2 Security and Functionality Goals* 122
- *4.4 Bellerophon Design* 124
  - *4.4.1 Preparing and Running a Binary* 124
  - *4.4.2 Scalable Key Management* 126
  - *4.4.3 Forward Security* 129
  - *4.4.4 Load-Balancing* 133
- *4.5 Security Analysis* 135
  - *4.5.1 Malicious Infrastructure Providers* 135
  - *4.5.2 Fully Malicious Users* 138
- *4.6 Case Study: Reusable Serverless Enclaves* 139
- *4.7 Performance Evaluation* 141

4.7.1 Verification Latency 143
4.7.2 Re-encryption and Rotation Latency 146
4.7.3 End-to-End Performance 147
4.8 Discussion and Limitations 149
4.9 Related Work 152
4.10 Conclusion 155

5 Conclusion and Future Work 156
5.1 Summary 156
5.2 Lessons Learned 157
5.3 Future Work 158
5.4 Closing Words 159

Bibliography 160

List of Tables

2.4 runsec API and Userspace Hypercall API 39
2.6 An overview of the lines of code, number of functions and third-party modules, and the languages of the target applications. . 47

3.2 Applet security requirements compared against various TEE-like systems. Keystone TEE based on RISC-V best fits our design requirements. 71
3.4 TypeScript subset that we compile in TAPDance. All variables are statically typed. 81
3.7 Interfaces in TAPDance. 96
3.10 Code Sizes of Applet Enclave Components and in TAPDance and the Node.js Interpreter 101
3.11 Performance Metrics of TAPDance. 101
3.12 Comparison of TAPDance with prior TAP security systems. TAPDance offers the best trade-off among all design criteria. . 106

4.5 Implementation size 142
4.9 Warm start performance of reusable serverless functions. The times for Intel ECDSA are in terms of the single-hop wide-area network latency, WAN. 148

List of Figures

2.1 An example of serverless application. In the application, the `UpdatesPhoto` function uploads the photo sent by an authenticated user to AWS S3, which triggers the `ProcessPhoto` function to generate a thumbnail and store the thumbnail into AWS DynamoDB. The rectangle with dashed lines represents compromised functions, and dashed lines are flow injected by the adversary. 22
2.2 An overview of Kalium architecture. 32
2.3 An example of the generated local flow graph for three traces ABBBCD, BABCD, and AFD. 37
2.5 Flow graphs of the target applications. Rectangles, rounded rectangles and dotted lines represent function, third-party services, and implicit flows, respectively. The entry functions that accept user requests are highlighted. 46
2.7 False-positive rates when using the traces collected from different numbers of rounds for building flow graphs. 50
2.8 The relative latency overhead of the functions in HelloRetail and the microbenchmark function. Valve proxies network requests only in functions marked with an asterisk. Y-axis is truncated at 5. 53

3.1 Example applet. 65
3.3 The TAPDance Architecture: Based on Keystone enclaves using RISC-V primitives. 76
3.5 Benchmarking Applet 85

3.6 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. 86
3.8 Example Benchmarking Applet 97
3.9 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. 100

4.1 Structure of interactive remote attestation: (1) The TEE-enabled machine sends the verifier a quote containing a proof of the initial user TEE state. (2) The verifier forwards the proof to the trusted hardware manufacturer, which (3) responds with the proof's validity. (4) If the proof is valid, the verifier provisions secrets into the TEE. 117
4.2 Binary preparation protocol 124
4.3 Decryption enclave protocol 125
4.4 The protocol for a hardware provider $\mathcal{H}$ to provision machine M owned by cloud provider $\mathcal{P}$. On a request to provision M with $id = [cpu, fwver, vk_{\mathcal{P}}]$, $\mathcal{H}$ verifies that M has CPU identifer *cpu* and firmware version *tcb* using the existing Intel EPID verification protocol. Before provisioning the key, $\mathcal{P}$ must authorize M through a simple challenge-response. On confirmation, $\mathcal{H}$ derives the HIBE decryption key for *id* using DeriveSKey and provisions it to M. 128

4.6 Intel EPID remote attestation time increases linearly with link latency. Bellerophon's non-interactive nature means launch does not depend on the link latency. Even for 10k page binaries, Bellerophon's entire launch process (which comprises Intel SGX enclave creation and the Bellerophon Decryption Protocol) is faster than EPID attestation (including the enclave creation time) 144
4.7 Bellerophon Decryption Protocol and Intel SGX enclave creation latency. Note that the Bellerophon Decryption Protocol is executed after Intel SGX enclave creation. 145
4.8 Impact of identity hierarchy depth on authenticator reencryption and minor epoch key rotation. 147

Abstract

Modern cloud providers offer a myriad of execution models defined by the dynamic computational resources available to the application. These execution models are primarily optimized for efficient resource utilization of datacenter resources to leverage economies of scale. In this thesis, we provide some preliminary evidence on how cloud infrastructure and applications can be co-designed to enable applications satisfying a greater number of security properties while preserving their scalability and performance characteristics. To guide this exploration, we select a subset of prominent cloud execution models and build performant systems with augmented security properties, namely Kalium, TAPDance and Bellerophon.

In the first part of the dissertation, we define a control flow integrity framework called Kalium. Kalium is an extensible security framework that leverages local function state and global application state to enforce control-flow integrity (CFI) in serverless applications. We evaluate the performance overhead and security of Kalium using realistic open-source applications; our results show that Kalium mitigates several classes of attacks with relatively low performance overhead and outperforms the state-of-the-art serverless information flow protection systems.

In the second part, we re-architect Trigger Action Platforms (TAPs), a type of distributed event processing system, 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.

Lastly, we present *Bellerophon*, a new remote attestation mechanism that eliminates the need for both trusted verifiers and separate secret provisioning in trusted execution environments (TEEs) deployed in cloud environments. TEEs form an execution model where user programs run in isolation and free from interference from all the system software running on the machine. Bellerophon accomplishes this feat using encrypted binaries embedded with user secrets that can only be decrypted using a manufacturer-provisioned key and only when the TEE is correctly initialized. Moreover, Bellerophon seamlessly integrates with existing approaches to accelerate confidential serverless functions designed to reduce launch overheads. Bellerophon uses Hierarchical Identity-Based Encryption (HIBE) to simplify secret key management and public key distribution, and incorporates a key rotation mechanism for forward security. Finally, our evaluation shows that Bellerophon provides similar security to existing interactive attestation mechanisms with much lower latency.

1 Introduction

The proliferation of cloud computing for outsourcing computations has made it a prime target for cyber attacks, primarily to extract or disrupt business value in the data and computations themselves. More specifically, applications deployed on the cloud have come under attack from external threat actors as well as insider actors, such as cloud provider employees whose work machines have been compromised by malware. The problem is twofold, cloud applications have become large and complicated [46], making them prone to bugs, and even if the application itself is bug-free, insider threats can undermine application and data security. In 2024, the average total cost of a data breach was $4.8 million, highlighting the ever-increasing need to solve this challenge [80].

The modern cloud is heavily optimized for efficient resource utilization and high performance owing to economies of scale when running large datacenters. In this thesis, we explore how cloud infrastructure and applications can be co-designed to enable applications satisfying a greater number of security properties while preserving their scalability and performance characteristics. In the following paragraphs, we describe cloud execution models and their historic relevance. We then explore modern cloud execution models, including their security challenges, and describe our contributions in addressing the challenges.

Programming distributed systems for efficient resource utilization has always been a challenging problem due to a variety of reasons ranging from fault tolerance to heterogeneity of infrastructure. Starting from the 1960s, there have been several efforts to build operating systems for distributed systems which can be used by users to run programs. Some examples include Sprite [137], the Cambridge Distributed Computing System [134], Plan 9 [142], Amoeba [131], V [35], and Eden operating systems [115]. Most of these systems were built during the early days of the Internet and

were designed to share the resources of a pool of networked machines, owned by a university or corporation by a set of employees. Each of these systems specify an execution model for user applications, which defines how computational resources (compute, storage and network) are accessed and multiplexed between various users. For example, the Cambridge Computing System assigned each user an entire physical machine with the requested specifications whereas Plan 9 supported migrating processes between different machines.

Similar to distributed operating systems, efficient multiplexing of computational resources over the public Internet has led to the rapid growth of the cloud computing industry, enabling users to achieve their computing goals at a fraction of the cost of owning machines themselves. Since the introduction of the virtual machine at IBM in the early 1970s [79], modern cloud and hardware providers have introduced several new techniques to multiplex computing resources, optimizing for performance, cost, and higher assurance against insider threats. Users are provided with abstractions of resources that can be used to build large, distributed applications. Each of these abstractions provide different guarantees, for example, a serverless function defines a limited unit of CPU time and memory that can be used to run relatively short computations, with the added guarantee of spawning additional copies of the function should the number of requests to the function exceeds a certain threshold (auto-scaling).

All of these techniques require cloud applications to be re-designed to take advantage of the added functionality, forming an execution model defined by the application's accessible resources and environment. These new execution models cater to a diverse range of requirements for modern-day outsourced applications, including faster scalability and better resource management [172], and newer threat models for security [7, 42, 91].

1.1 Execution Models

A cloud execution model is defined by the organization of compute, storage, and communication resources that are accessible to run user applications. With the key selling point behind cloud computing being resource sharing, the execution models define the flexibility of dynamic resource allocation. Given an execution model, the user can tailor applications depending on the end goal, whether it be better resource utilization, scalability or security.

In this thesis, we focus on the following modern cloud execution models:

Function as a Service (FaaS): Serverless computing (or function-as-a-service, FaaS) is an emerging application deployment architecture that completely hides server management from tenants. Serverless has a new programming model: an application is decomposed into small components, called *functions*, each of which is a small application dedicated to specific tasks that runs in a dedicated *function instance* (a container or another kind of sandbox) with restricted resources such as CPU time and memory. A function instance, unlike a virtual machine (VM), will be launched only when there are requests for the function to process and is paused immediately after handling one request. Serverless has been used as a general programming model for a variety of applications [64, 109, 204].

Trigger Action Platforms: Trigger-action platforms (TAPs) enable end-users to automate interactions between a wide variety of third-party online services and devices [107]. 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 [85], Zapier [209], and Microsoft Power

Automate [123]. Trigger-action platforms have been implemented using a lower level execution model such as FaaS [130].

Trusted Execution Environments: Trusted execution environments (TEEs) allow clients to run code securely in a harsh environment: machines controlled by an attacker running malicious system software. Their widespread deployment in recent years has drastically reduced the cost of secure computing in domains including banking [63, 125], health care [93, 94, 95], and blockchains [44, 135, 136]. Major cloud providers, including Amazon, Google and Microsoft, now provide TEE-enabled compute resources [124] based on hardware, including Intel SGX and TDX and AMD SeV-SNP [8, 90, 96].

These new execution models present unique challenges and opportunities in the face of rapidly increasing cyber threats against cloud infrastructure.

1.2 Security Challenges and Opportunities in Existing Cloud Execution Models

Function as a Service (FaaS). The growing adoption of serverless computing gives rise to new security challenges. Like conventional web applications, vulnerabilities in the functions or third-party libraries being used can be exploited by adversaries to subvert the control flow and data flow of applications, in order to steal sensitive data and perform stealthy operations, as demonstrated in [154, 157, 159]. Existing security tools for web applications have been ported to serverless applications, such as vulnerability scanning tools and log-based anomaly detection but they are only useful for detecting known vulnerabilities, or for non real-time attack detection[13, 145, 181].

Previous work leverages information flow control (IFC) [5, 47] to solve the problem of **serverless data confidentiality** (See §2.2 for the definition)

in serverless applications. However, IFC based techniques have not been explored in the context of serverless applications to solve the problem of **serverless data integrity** (§2.2), i.e., preventing an unauthorized user making changes to stored data. At a high level, existing information flow control systems do not enforce the intended order of execution of functions in the serverless application, the violation of which causes the application as a whole to be insecure and may allow an adversary to modify data in an external store. For example, an adversary may bypass an authentication function to directly invoke a function that has write access to a datastore (see §2.3.1).

Trigger Action Platforms (TAPs). As TAPs are large-scale systems with millions of users [107], they become centralized hubs with privileged access to user data and devices (e.g., the popular IFTTT platform boasts 20 million users [87]). 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 [2, 76, 129, 191]. 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 [83]. Consequently, some third-party services (e.g., GMail) have become reluctant to interface with TAPs citing privacy concerns [84]. At the same time, users are becoming wary of the insufficient safeguarding of their data by TAPs [50]. 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 [34, 37, 58, 129, 132, 166, 207, 208]. 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.

Trusted Execution Environments. TEEs are extremely powerful, but with the TEE enclave itself entirely surrounded by an untrusted operating system, untrusted facilities, and an untrusted network, there is more work to do. To launch an enclave, a client simply sends a piece of code to the cloud provider, which runs it inside a TEE. Since the code passes through the untrusted system, it cannot contain secrets, and before sending secrets, even in encrypted form, a client must verify that (1) the correct code is running, and (2) it is running in a valid enclave that untrusted software cannot pierce, lest an attacker maliciously modify the code or attempt to run it where they can observe its data. Existing TEEs accomplish these goals through *remote attestation*, a protocol that allows a client to verify that the enclave was initialized in the correct state and is running on a

genuine TEE-enabled CPU. There are currently two approaches to remote attestation, both with substantial drawbacks.

Traditional attestation, defined by hardware manufacturers like Intel [92, 98] and AMD [9], places the burden of verification fully on the client. The client must receive an attestation from the enclave and verify it with the hardware manufacturer. This model provides very strong security, guarding against an adversarial cloud provider, but clients must run their own verifiers and implement their own secret management, deciding when and how to transmit secrets to a verified enclave. A commercial grade secret management service such as HashiCorp Vault [75] could be as large as 650K lines of code. Moreover, clients must communicate with each enclave on launch (and to the hardware manufacturer in some attestation schemes), not only requiring them to be online and available, but adding wide-area network latency to the process. This extra latency alone can be prohibitive in applications that require rapid processing of large data, like serverless and high-performance compute workloads. Prior work on protecting serverless functions inside TEEs rely on the client attesting the enclave each time a function is invoked [69, 146, 175] (including the commercial Conclave Cloud Functions [25]). Our measurements show that with a wide-area network latency of 10 ms, attestation increases startup latency 40-400% compared to published AWS cold-start latencies [18].

To solve these scalability and performance problems, commercial deployments, such as those at Microsoft [127] and Amazon [19], place immense trust in the cloud provider. Instead of the client performing their own verification and secret management, the cloud provider does both. Users no longer have to provision secrets to enclaves, and launch avoids wide-area network communication as the cloud provider can place all relevant servers in the same datacenter. However, users must now completely trust the complex services of the cloud provider, reducing the

value of TEEs. Intel recently launched a dedicated cloud-based, distributed attestation service called Trust Authority [102] designed to reduce the burden of verification on the users by supporting the offloading of the user's TEE launch policy. In this model, the user still needs to manage secrets and transmit them to the TEE after a successful evaluation of the launch policy by the Trust Authority, and the Trust Authority is a large addition to the user's TCB. Prior work has also used verifiers running inside enclaves [65] running in the same datacenter, removing the infrastructure provider from the TCB. However, managing the state of verifier enclaves poses the same management challenges as the client running their own verifier, increasing the overall TCB size.

Non-interactive attestation, which does not require online verification, is a promising solution. Previous work on non-interactive remote attestation focused on either deferring the verification of the proof of initial state or relying on periodic verification of the initial state by the remote verifier [108, 120, 173] but does not consider secret provisioning. Furthermore, launch still requires one round trip with the verifier, which is still part of the TCB. Chancel [3] runs a loader enclave that is attested in advance and is entrusted with the secrets needed to decrypt the binaries input by the client. While this approach alleviates the impact of attestation on startup latency, the burden of managing the loader enclaves and provisioning secrets still falls on the client, increasing the size of their TCB.

1.3 Contributions

1.3.1 Defining and Enforcing Control Flow Integrity for Serverless Applications

Information Flow Control (IFC) based techniques have not been explored in the context of serverless applications to solve the problem of **serverless**

data integrity (§2.2), i.e., preventing an unauthorized user from making changes to stored data. We would like to complement IFC-based approaches with our techniques for data integrity protection. Several common design patterns of serverless applications can be leveraged to improve serverless security: (1) Tenants need to externalize the data produced by the function to other services for later use, to avoid data loss due to the *stateless* nature of serverless functions. Such externalization behaviors can be monitored and used for anomaly detection. (2) As complex applications are decomposed into dedicated task functions with relatively simple logic, it is possible to model each function individually and construct a global view of the application. (3) Decomposition of an application makes it easier to enforce customized policies for different components, thereby facilitating more flexible and efficient security monitors.

Inspired by the above insights, we design a novel serverless security framework that we call *Kalium*. Kalium enforces control-flow integrity (CFI) for each individual function in the serverless application as well as for the application (which may be composed of multiple functions) as a whole. Kalium provides data integrity, which is complementary to data confidentiality. Unlike conventional CFI (e.g., [1]) that enforces a predetermined control flow graph of a program based on the instruction pointer values, we treat the network messages made by a serverless function (or the application as a whole) as the edges in a per-function or application-wide control flow graph. Each function in a serverless application is modeled using a function control flow graph which captures the order of various interactions of the function with external services, ending with a control transfer to another function. The serverless application as a whole is modeled using a global control flow graph which captures the control flows between different functions.

In Kalium, a function runs in a modified container runtime environment called runsec that intercepts certain system calls (e.g., `SendMsg` and `Write`) and passes current function state to a *guard* module. The guard keeps track of the local control flow graph of the function, checks whether a network system call should be allowed or not and returns the expected action that will be enforced by the runtime. A per-application *controller* centralizes the tracking of the application-wide (global) control flow. It coordinates and collects function states from guards, and uses the global state to help the guard make decisions during inter-function and external control flow transfers. Control flow graphs (local and global) are built using semi-automated analysis of the control paths in the application.

The runsec runtime is built atop gVisor, a secure container runtime [152]. We extend gVisor to support a set of APIs that (1) allow the guard to block system calls based on high-level information such as network request payload and URL, and (2) allow functions to offload encrypting network traffic (e.g., for TLS) to runsec to facilitate inspection and modification of the payload before being sent out. Our methodology not only reduces the attack surface of gVisor-based containerized applications, but also facilitates the flexible control of application behaviors, which can serve as a building block for future gVisor-based security applications. We will publicly release our extensions.

We implement a prototype of Kalium, and evaluate the performance overheads of Kalium and the security of flow tracking with various applications and workloads. Our analysis shows that Kalium introduces relatively small overhead while preventing attacks that are not handled by existing serverless security tools. Kalium was presented in the 32nd USENIX Security Symposium.

1.3.2 Architecting Trigger Action Platforms using Trusted Execution Environments for Security, Performance and Functionality

We contribute to the TAP 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 3.9 contains a comparative analysis). Specifically, our work achieves better performance and functionality than eTAP [34] and Walnut [166] while offering similar security (under different assumptions), and stronger security than minTAP [207], DTAP [58] and oTAP [37]. 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 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 [34] or Walnut [166]. At a 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 3.7.1, 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 [7, 48, 59, 91]. 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 3.4.2). 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.

TAPDance was presented in the ISOC Network and Distributed Systems Security Symposium 2024 (NDSS '24).

1.3.3 Non-Interactive Remote Attestation

We present Bellerophon, an attestation scheme with the scalability and performance of provider-managed verification and secret management, the traditional threat model trusting the cloud provider only to not deny service, and a smaller trusted computing base (TCB) than prior approaches. The key insight enabling these gains is to encrypt user binaries such that only the TEE inside a genuine and properly-provisioned CPU can correctly decrypt them. As a result, clients can safely include secrets in their initial (encrypted) binary, and be sure *without interactive verification* that only a secure enclave can decrypt them. This assurance eliminates the choice between high trust in the cloud provider and online verification with large network latencies. In addition, it allows binaries

to be stored and executed asynchronously, even behind firewalls, at any time at low cost. To maintain compatibility with existing machines capable of running TEEs, Bellerophon does not require any changes to existing cloud machine hardware capable of running TEEs but requires the hardware manufacturer to update their protocols. Each Bellerophon worker machine requires a persistent and tamper-proof key store that is protected from untrusted software, so we assume a hardware Trusted Platform Module (TPM) in our design.

Designing an encryption-based attestation scheme presents important challenges. First, the successful decryption of the user's binary must depend on the TEE being loaded correctly, just as today's TEE's rely on remote verification to run successfully. This requires great care in the design and, as we discuss in Section 4.4, is not scalable using a standard public-key encryption scheme. Second, as with any protocol sending encrypted secrets through an untrusted channel—in this case a cloud provider—the scheme should be forward secure. That is, a compromise of key material should not leak secrets from prior to the compromise, even if the attacker stored previously encrypted data.

We implement a Bellerophon prototype on Intel SGX enclaves, and show that the Bellerophon's TCB is small: just over 1,000 lines of code for a local architectural enclave for decryption, and 38.6K lines in total including all the other local architectural enclaves, library code and the provisioning server. The addition of a hardware TPM into the TCB is in line with the usage of TPMs for VM attestation in prior academic work and real cloud deployments [128, 133, 188]. Our experiments compare Bellerophon's performance against SGX-style interactive attestation (also applicable to virtual machine enclaves such as Intel TDX [96] and AMD SeV-SNP [8]) and show that Bellerophon reduces enclave startup latency for a 40 MB TEE binary by 94% when the SGX verifier requires a 10ms network round trip. To demonstrate the effectiveness of Bellerophon in an end-

to-end application, we integrate Bellerophon with the reusable enclaves framework [175] to accelerate the startup performance of confidential serverless functions.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 describes Kalium, a control flow integrity framework for serverless applications. Chapter 3 describes TAPDance, an alternate trigger-action platform architecture that achieves a better trade-off between security, performance, and functionality compared to prior work. Chapter 4 introduces Bellerophon, a non-interactive and verifier-free remote attestation scheme. Finally, we conclude in Chapter 5 with a discussion of future research directions.

2 Control Flow Integrity for Serverless Applications

2.1 Introduction

The growing adoption of serverless computing gives rise to new security challenges. Like conventional web applications, vulnerabilities in the functions or third-party libraries being used can be exploited by adversaries to subvert the control flow and data flow of applications, in order to steal sensitive data and perform stealthy operations, as demonstrated in [154, 157, 159]. Existing security tools for web applications have been ported to serverless applications, such as vulnerability scanning tools and log-based anomaly detection but they are only useful for detecting known vulnerabilities, or for non real-time attack detection [13, 145, 181].

In this work, we define **serverless data integrity** (§2.2) and show how it can be enforced. Previous work leverages information flow control (IFC) [5, 47] to solve the problem of **serverless data confidentiality** (See §2.2 for the definition) in serverless applications. However, IFC based techniques has not been explored in the context of serverless applications to solve the problem of **serverless data integrity**, i.e., preventing an unauthorized user making changes to stored data. At a high level, existing information flow control systems do not enforce the intended order of execution of functions in the serverless application, the violation of which causes the application as a whole to be insecure and may allow an adversary to modify data in an external store. For example, an adversary may bypass an authentication function to directly invoke a function that has write access to a datastore.

Several common design patterns of serverless applications can be leveraged to improve serverless security which complements existing IFC based techniques: (1) Tenants need to externalize the data produced by

the function to other services for later use, to avoid data loss due to the *stateless* nature of serverless functions. Such externalization behaviors can be monitored and used for anomaly detection. (2) As complex applications are decomposed into dedicated task functions with relatively simple logic, it is possible to model each function individually and construct a global view of the application. (3) Decomposition of an application makes it easier to enforce customized policies for different components, thereby facilitating more flexible and efficient security monitors.

Inspired by the above insights, we design a novel serverless security framework that we call *Kalium*. Kalium enforces control-flow integrity (CFI) for each individual function in the serverless application as well as for the application (which may be composed of multiple functions) as a whole. Kalium provides data integrity, which is complementary to data confidentiality. Unlike conventional CFI (e.g., [1]) that enforces a predetermined control flow graph of a program based on the instruction pointer values, we treat the network messages made by a serverless function (or the application as a whole) as the edges in a per-function or application-wide control flow graph. Each function in a serverless application is modeled using a function control flow graph which captures the order of various interactions of the function with external services, ending with a control transfer to another function. The serverless application as a whole is modeled using a global control flow graph which captures the control flows between different functions.

In Kalium, a function runs in a modified container runtime environment called `runsec` that intercepts certain system calls (e.g., `SendMsg` and `Write`) and passes current function state to a *guard* module. The guard keeps track of the local control flow graph of the function, checks whether a network system call should be allowed or not and returns the expected action that will be enforced by the runtime. A per-application *controller* centralizes the tracking of the application-wide (global) control flow. It

coordinates and collects function states from guards, and uses the global state to help the guard make decisions during inter-function and external control flow transfers. Control flow graphs (local and global) are built using semi-automated analysis of the control paths in the application.

The runsec runtime is built atop gVisor, a secure container runtime [152]. We extend gVisor to support a set of APIs that (1) allow the guard to block system calls based on high-level information such as network request payload and URL, and (2) allow functions to offload encrypting network traffic (e.g., for TLS) to runsec to facilitate inspection and modification of the payload before being sent out. Our methodology not only reduces the attack surface of gVisor-based containerized applications, but also facilitates the flexible control of application behaviors, which can serve as a building block for future gVisor-based security applications.

We implement a prototype of Kalium, and evaluate the performance overheads of Kalium and the security of flow tracking with various applications and workloads. Our analysis shows that Kalium introduces relatively small overhead while preventing attacks that are not handled by existing serverless security tools.

Our work presents the following contributions:

- Design and open-source implementation of a flexible, extensible serverless security framework called Kalium, which allows for control flow protection in serverless applications, with a novel encrypted traffic interception technique achieved by offloading encryption into the container runtime.

- Identification of serverless-specific challenges in control flow monitoring, and design of mechanisms for control flow modeling and enforcement that can track control flows across services.

- Evaluation and comparison of performance overhead with existing work.

2.2 Background

Kalium enforces control-flow integrity, and in this section, we define the serverless application and function control graphs. We also introduce information flow control, which is later used to show the strengths of Kalium in protecting the integrity of serverless data.

2.2.1 Serverless data confidentiality and integrity

Data confidentiality for a serverless application means that all data that are being used/computed during application runtime shall not be leaked to an unauthorized output channel (e.g., a remote party and database). Whether data computed from a particular input source could be exposed to a particular output source is specified by the application developer. In the context of serverless applications, an input source to a function is either an external service or another function. An output channel could be an external service. An example of data confidentiality policy is "data from the user database should be exposed only to select external services". The challenge in maintaining data confidentiality is that each use of a sensitive data item should be tracked throughout the runtime of the serverless application.

The complementary problem to data confidentiality is data integrity. Data integrity means that an unauthorized user should not be able to modify data or alter the state stored in an external data store, meaning that data should not be allowed to flow from an inappropriate location. In the context of serverless applications, an inappropriate location is a function that does not satisfy a precondition before its execution. An example is a function that writes to a database and is executed without a preceding execution of its caller-authentication precondition function. Enforcing data confidentiality does not necessarily enforce data integrity as we explain in §2.3.

To summarize, serverless data confidentiality and integrity are **complementary goals**. In our work, we show how Kalium can be used to protect serverless data integrity.

2.2.2 Serverless control flow

For our work, we define two types of control flow graph (CFG) for serverless applications: (1) application control flow and (2) function control flow. We define the application control flow graph of serverless applications as a directed graph $G = (V, E, s, F)$ where each of the vertices $v \in V$ is a function and $s \in V$ is the starting vertex. F represents the set of ending vertices. A directed edge $e \in E$ between two vertices v_i and v_j represents a message passed between the two nodes either directly or through an external service such as a message queue. We call edges that represent messages passed through an external service as *indirect flows*. An ending vertex does not pass messages to another vertex. For our work, we assume messages are passed in HTTP format.

We define the function control flow graph of a serverless function f as a directed graph $G_f = (V_f, E_f, s_f, e_f)$ where each of the vertices $v \in V_f$ is the internal function state right before sending a message to an external service that does not send its response to a different function $f'(\neq f)$. In other words, the external service performs an action and returns its result to the invoking function f and does not transfer control to another function f' in the application. $s_f \in V$ denotes the start state while $e_f \in V$ denotes the end state. On reaching state e_f, the function (1) returns a value val or (2) calls an external service that invokes another function f' in the application or (3) directly calls another function f'.

G_f and G should be consistent to capture the actual application behavior that is if e_f transfers control to another function f' (directly or through an external service) then there must be an edge from v_f to $v_{f'}$ in G.

An edge exists between two vertices where there is control flow, and can also indicate the nature of the flow. For example, the existence of an edge between two vertices may also incorporate particular message metadata or contents (for ex. a URL parameter or POST request data). The resulting CFG should not yield **false negatives**, i.e, a violation of the true application behavior should always be reported by the CFG. The CFG may have **false positives**, i.e., expected application behavior that is flagged as a violation. The elimination of false negatives is a necessary condition for security because no attack should go undetected, whereas the elimination of false positives pertains to usability of the application.

2.2.3 Information flow control in serverless

Information flow control (IFC) labels the sources and sinks of data in an application to prevent information leakage. The labels form a lattice and represent security classes of information in the application. The labels are propagated during the application execution according to the rules of a particular scheme. Information obtained from an input source with a particular label l_0 can only be exposed to an output channel with label l_1 iff $l_0 \sqsubseteq l_1$, that is l_0 is lesser than or equal to l_1 in the partial-order defined by the security lattice.

Data integrity has been solved with IFC [165] for regular programs. However, using IFC for serverless data integrity has not been explored in previous work. Trapeze [5] is an IFC framework for serverless applications. However, it only provides the termination sensitive non interference property (TSNI), which can leak secrets by observing whether a function terminates or not. Valve [47] is a taint tracking (IFC with a specific type of lattice of labels) framework for serverless. Both Trapeze and Valve focus on protecting serverless data confidentiality.

We show concrete attacks in §2.3.4 that demonstrate that standard IFC as implemented does not solve the problem of serverless data integrity. In contrast, Kalium provides serverless data integrity, which is

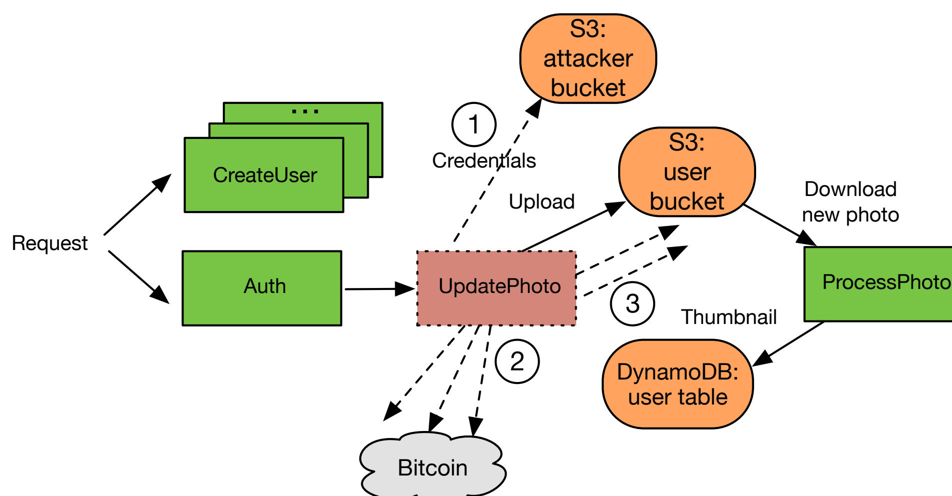

Figure 2.1: An example of serverless application. In the application, the `UpdatesPhoto` function uploads the photo sent by an authenticated user to AWS S3, which triggers the `ProcessPhoto` function to generate a thumbnail and store the thumbnail into AWS DynamoDB. The rectangle with dashed lines represents compromised functions, and dashed lines are flow injected by the adversary.

complementary to serverless data confidentiality. We envision that Kalium can be used along with IFC frameworks to provide protection against a bigger class of attacks.

2.3 Motivation and Challenges

In this section, we introduce different types of attacks against serverless applications, the limitations of existing serverless security tools, common design patterns of serverless applications that inspire our system design, and the challenges in developing efficient security mechanisms.

2.3.1 Attacks against serverless applications

Serverless design has drawn attention from adversaries and researchers, and several attacks, which mostly tamper with control flow, have been proposed. We illustrate an example application in Figure 2.1 and three types of control flow related attacks that motivate our system design:

(a) The adversary manipulates function execution order (i.e., application control flow) to subvert application logic, while function control flow is not affected. For example, the adversary directly invokes `UpdatePhoto` without being authenticated by `AUTH`. These types of attacks are not detected by existing IFC-based protections because they do not take into account the order in which functions are invoked, leading to the execution of the `UpdatePhoto` with the correct security label. However, the security label carries no information whether authentication succeeded in the `AUTH` function or not. (see §2.3.4).

(b) The adversary tampers with function control flow but does not affect application control flow by exploiting legitimate execution paths of the application. In Figure 2.1 (3), the adversary performs DoS attacks against `ProcessPhoto` by uploading a large amount of data from `UpdatesPhoto` to the S3 bucket. These types of attacks are not detected by IFC-based protections as security labels have no concept of counting the number of times an output channel is accessed. Hence even if the `UpdatesPhoto` function executes with the correct security label, it does not stop multiple messages from being placed on the output channel when only one is expected.

(c) The adversary hijacks both application and function control flow. Examples are **data exfiltration attacks** and **crypto-mining attacks** [26, 157, 159]. In data exfiltration attacks, the adversary exploits vulnerable functions, i.e., functions with vulnerabilities in the libraries or code, to steal sensitive data stored in the function code or environment variables. For example, in Figure 2.1 part (1), the adversary extracts the AWS credentials from the `UpdatesPhoto` function and sends them to her own S3 bucket. In crypto-mining attacks, as in Figure 2.1 (2), the adversary performs her activity stealthily at the cost of the tenant. A concrete attack proposed by PureSec [159] shows that the adversary can turn one single vulnerable function into a virtual crypto-mining farm without being noticed by the tenant. Further, a new malware sample called Denonia[26] that targets AWS Lambda to launch a variant of the XMRig [202] crypto-mining software has been discovered recently. SandTrap [130] shows how the IFTTT rule sandbox run on AWS Lambda can be bypassed to exfiltrate customer IoT event data. IFC-based protections can detect such attacks as they involve deciding whether information should flow to an adversary-controlled output channel.

In this work we demonstrate that Kalium can defeat all three of the above-mentioned attacks.

2.3.2 Common design patterns of serverless applications

To guide our development of defenses against the aforementioned attacks, we examined 50 open-source serverless applications (86 functions) on GitHub[1] to understand their common design patterns. Below we highlight three patterns that we believe should be considered when developing security-enhancing mechanisms for serverless applications.

- *No local storage.* In general, no examined applications store data (e.g., intermediate processing results) on local disks due to the *stateless* nature

[1]We conducted this survey before the release of the Wonderless serverless application dataset [55].

of serverless. Serverless providers do not guarantee that requests can always be handled by the same function. So, storing stateful data on local disk faces the risks of data loss. The common practice for passing data across requests is to store data on some storage service(s) and retrieve it later.

- *No direct interactions between functions.* There are usually no direct information flows between functions. Applications tend to rely on other services (e.g., event queues and storage services) to transfer control from one function to another function. Transferring data across functions is the same as across requests, relying on storage services. In fact, in AWS Lambda, the size of function input is limited to 6 MB (for synchronous requests) or 256 KB (for asynchronous requests), which may not be sufficient for some applications. To safely transfer arbitrary data between functions or requests, applications use storage services.
- *Input-dependent functions are common.* The number of requests and their target URLs may depend on input parameters. For instance, a weather application fetches weather information for all the cities mentioned in the input from an API service that uses different URLs for different cities, and sends back an aggregated result. Similarly, an application may only set up one serverless function as an entry point, and performs different operations based on user input (i.e., like a dispatcher or switch statement); in fact, this is the fastest way to port conventional web applications (e.g., Django-based web applications) to serverless.

HTTPS requests from a function indicate either a data transfer or a control transfer. By monitoring HTTPS requests within an application, we can therefore monitor the control flow of the application. For most of the functions, the destinations of their requests are known, which means we are able to model their behaviors. For some functions, we can only know the requested URLs at runtime. However, one useful observation is that the URLs requested by such functions have a fixed pattern, mostly in the

form of “common prefix + variable”. We only consider this pattern in our project, though there could be other patterns.

2.3.3 Challenges

To prevent the attacks discussed in §2.3.1 we need to accurately track the control flow of an application. Existing information flow protection mechanisms usually monitor system calls of interest by modifying OS or system libraries [140, 211]. Such mechanisms often assume that an endpoint can be identified by IP and port, which is not sufficient for serverless functions. In serverless design, a function is associated with dynamically assigned and ever-changing IPs, and a service might have the same IP and port as other services (e.g., one can redirect requests to different services on the same host based on the `Server Name Indication` field). So, the URL is necessary for endpoint identification, but it is difficult to directly extract high-level information such as URL and HTTP header from the low-level information seen by such mechanisms. Considering the limited resources in function instances, we need to wisely choose the granularity of monitoring to reduce monitoring overhead and capture more meaningful information at the same time.

Furthermore, serverless functions may be input-dependent, i.e., number of requests and endpoint URLs vary based on input parameters, making them challenging to model.

Another challenge is tracking control flow across different external services. As mentioned before, a serverless application often relies on functionality provided by third-party services, such as hosted databases. It is unrealistic to assume that all the services are willing to upgrade their infrastructures to support new security mechanisms.

Finally, for ease of development, a new security mechanism should be application-independent and transparent to applications, i.e., application code does not need to be modified.

2.3.4 Limitations of existing approaches

Information flow control

IFC-based techniques for serverless data integrity have not yet been explored in previous work. Currently, IFC-based techniques[5, 47] can ensure data confidentiality, that is information labeled as high security will never be leaked to an output channel labeled with a lower security label. Consider a sub-path in the application graph $P = v_i...v_j...v_k$ where node v_j authenticates the caller of node v_i and the execution of v_k should be allowed only if the execution of v_j succeeds. Each node inherits the security label of its caller. If after the execution of P (after successful authentication at v_j), node v_k has label l_1. Now suppose an adversary compromises v_i and directly calls v_k, bypassing the authentication at node v_j. In this case let the label of v_j be l_0. Then $l_0 \sqsubseteq l_1$ because a label can only be higher up in the lattice as more labels are accumulated. This implies that the adversary has enough privileges to cause node v_k to execute all the actions that it would have (for example: modifying a user database), had the authentication succeeded at node v_j. This example can be extended to cases where the single node v_j can be replaced with a series of nodes, $v_{j_1}...v_{j_n}$ that need to be executed as a precondition before the execution of node v_k. In such cases, it is hard to decide whether node v_k should be executed or not without tracking the exact path that was executed prior to the node v_k. Hence, current IFC-based techniques (and IAM rules) do not suffice for ensuring serverless data integrity.

To the best of our knowledge, IFC-based techniques cannot prevent multiple executions of the same sub-path in a program due to the fact that *security labels are not order preserving*. Consider a node v_i in the serverless application graph that has an edge to a node v_j. If node v_i is compromised, then the adversary can issue multiple identical requests to node v_j with the same security label as that of node v_i, potentially leading to multiple

executions of one or more application sub-paths starting with node v_j. If the execution of a particular sub-path does not result in an idempotent operation being performed, then the application semantics can be altered by multiple executions of the same sub-path. A concrete example is a banking application where the execution of a particular sub-path results in the bank balance of an account to increase by some amount. An adversary can leverage this to accumulate funds in an account by executing that sub-path multiple times.

Allow list based policies

Allow list based tools [39, 103, 145, 150, 153, 160] usually implement simplified information flow control by running functions in a sandbox and let tenants specify and control the resources a function can or cannot access. However, such policies cannot detect manipulation of legitimate control flows, e.g., out-of-order or repeated control flows. In addition, they focus on securing each individual function and ignore the specific nature of serverless applications. The lack of visibility into the entire application causes these tools to fail to detect attacks that leverage incorrect function execution order (i.e., invalid application execution paths) to subvert application logic and violate data integrity.

To summarize, similar to IFC-based techniques, allow lists do not keep track of the path of the serverless application being executed, and so they cannot prevent the attacks described in §2.3.4.

Log analysis techniques

Host-based intrusion detection systems (HIDS) detect potential attacks by monitoring an application's execution [49]. Model-based HIDS, a specific type of HIDS, builds a model of the expected execution behavior (i.e., allowed sequences of system calls) of the monitored application, and compares the system calls issued by the application during its execution against the model to detect anomalies [62]. The model is usually

represented by an automaton. There is model-based HIDS research, focusing on model construction (dynamic analysis [57], static code analysis [195], static binary analysis [67], etc.) and model design (abstract stack model [195], Dyck model [106], inlining model [71], etc.). ALASTOR [144] is a provenance framework for serverless applications that provides fine-grained tracking of application behavior using logs from various sources, including system call tracing and network profiling. Control-flow integrity (CFI) can be treated as a special type of intrusion detection mechanism for enforcing a nondeterministic finite automaton (control flow graph) to prevent the application from deviating from normal execution paths [1]. Our work is inspired by these works and applies model-based intrusion detection and CFI to the new setting of serverless applications. To the best of our knowledge, serverless data integrity has not been a specific target so far for HID systems.

2.4 Threat Model

We consider three major parties in our threat model: a target application, an adversary, and third-party services. The target application is deployed on a serverless computing platform by a trustworthy tenant (the application owner). An application might consist of multiple functions. Each function is assumed to be implemented as a single-threaded process within its container. Any services/applications/functions, other than the functions in the target application, are considered to be *third parties*, including services from the same cloud provider or set up by the same tenant outside the serverless platform.

We treat a third-party service as a blackbox that takes input from some sources and may output results to some destinations. Both the input sources and output targets (if any) of the service must be within the target application. We currently cannot enforce control over flows if the destination of the third-party service is not an in-application function. The exact functions that generate an input and receive the corresponding output might be different, though. For example, as in Figure 2.1, `UpdatePhoto` uploads a picture (input) to S3, and S3 will generate an "upload" event (output) to trigger `ProcessPhoto` to process the picture.

The application may store sensitive authentication data such as encryption keys or access tokens within the function's code and, therefore within each function instance.

Adversary capabilities We assume that the adversary can compromise at least one function of the application, leveraging bugs in the function code, vulnerable libraries used in the functions, or inappropriate configurations. The platform itself is assumed to be secure. By that we mean that the adversary cannot compromise host VMs, serverless runtime, third-party services or manipulate network traffic within the

deployment infrastructure. Attacks that leverage side channels in network communication such as timing-based attacks (e.g., [117]) are beyond the scope of this work. An adversary may also attempt to exploit spurious flows in the flow graph that is deployed for a compromised function. We discuss more about flow graphs in §2.5.1.

Adversary goals The adversary seeks to perform any type of control flow related attacks discussed in §2.3. We further assume that all the operations the adversary can perform must be done via the functions.

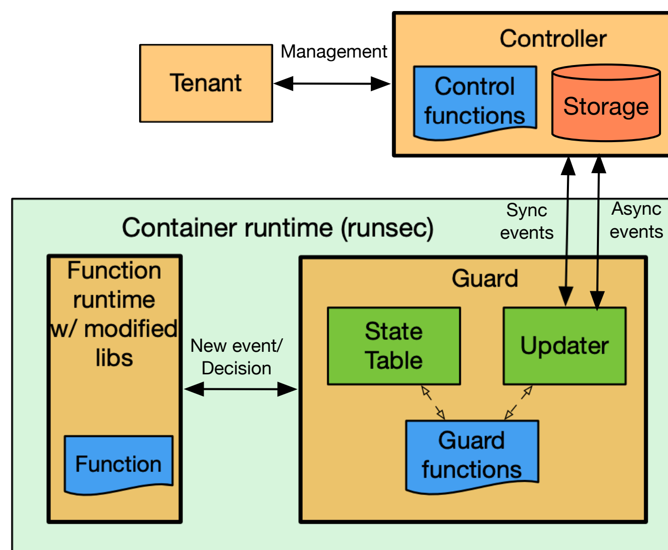

Figure 2.2: An overview of Kalium architecture.

2.5 Design of Kalium

Motivated by the above discussion, we design a novel, extensible system for protecting control flow in serverless application that we call *Kalium*. As shown in Figure 2.2, Kalium consists of two basic components: secure runtime and controller. The secure runtime consists of a function runtime and a *guard* module which tracks and enforces the local function control flow graph. The controller tracks the global (application-wide) control flow graph and helps the *guard* to make decisions during function-to-function and external network requests. We discuss flow graph generation and the guard and controller in this section.

2.5.1 Control flow tracking

All of the previously discussed attacks in §2.3 alter the control flow of the application as a whole or an individual function. To enforce Control-Flow Integrity (CFI) in serverless applications, we borrow the ideas from

model-based IDS for web applications and the original CFI technique [1, 67, 106, 195]. For application control flow, we treat an application as a single program and each of its functions as a basic block in the original CFI [1], and add checks when a function receives requests and returns responses. For function control flow, we built a model of the acceptable request sequences of a function, and monitor the requests sent by the function. We define a *flow* as one message exchange between a function and another endpoint (a service or a function).

We introduce two types of flow graphs for an application: global graph and local function graph. A global graph is a directed graph, where a node represents a function, and an edge from node A to node B (denoted by $A \to B$) represents that function A sends messages (directly or indirectly) to function B from connections initialized by A. Each function is associated with a local graph, wherein a node represents a network operation performed (e.g., an HTTPS request) by the function, and an edge points to the next expected operation.

Kalium can be extended with traditional CFI [1] in conjunction with the local control flow graph. Since traditional CFI is local to the function, the construction of the global graph remains the same. Kalium does not adopt traditional CFI directly because CFI for monolithic programs does not take into account the arguments to network function calls (e.g., URL).

Overview. We assume all the functions support Kalium while services do not. Let f be a function, m_{in} be the message f received from the standard function entry point, and m_{out} be the message f returns via the standard exit point. Note that f can only receive one m_{in} and send one m_{out} in an execution. m_{req} is a message sent from f to another endpoint s, and m_{resp} is the response from s. m_{req} and m_{resp} are sent over the same connection. We currently are not aware of any serverless platform that supports direct access to functions via their IP addresses, so m_{in} and m_{resp} correspond with the only two ways for passing messages to functions. Similar to CFI for

regular programs [1], the serverless environment including the runtime is responsible for adding an identifier, called the *tag* to each message identifying the sender of the message. External services are expected to propagate the tags in case they call a function.

The semi-automated method that we describe in this section can automatically generate *flow graphs* for a target application. The generated flow graph is a variant of a flow-sensitive and context-sensitive nondeterministic finite automaton (NFA) that models the expected flow sequences of a function or valid execution paths of an application. To construct such NFAs, we trade off space for model accuracy by duplicating states and removing cycles from NFAs as in the IAM model [71]. Next, we discuss how to generate these graphs.

Generating control flow graphs

Kalium collects execution traces of serverless application and provides a tool to create the flow graphs from the collected traces. A trace is the sequence of flows generated by all the functions (in the application) in one *application* execution, and records important information such as timestamps, function names, destination URLs, and HTTP operations (`GET`, `POST`, etc.). Generating accurate flow graphs that precisely cover all control flows is a challenging problem. The user could follow the best-effort strategy proposed by prior work [47], i.e., running Kalium in logging-only mode to collect traces under real workloads and then iteratively refining the graphs manually. In our work, we focus on defining and enforcing flow graphs, and leave automated generation of precise flow graphs as future work.

Given a set of traces, Kalium leverages the method proposed in Synoptic [23], which is originally designed for building loop-free NFAs from syscall traces, to generate flow graphs. In the local graph, each node is a $\langle$URL, HTTP operation$\rangle$ tuple that indicates that the function sends a message to an endpoint whose address is *URL*. A tag is an identifier to

track the caller of a function (similar to labels in the original CFI) and has the format of ⟨function name, guard ID, request ID⟩, while the request ID is just a random 16-byte string assigned to each request, generated by the entry function of the application. The local graph has an entry node indicating the endpoint from whom f receives m_{in}, and an exit node indicating sending out m_{out}. In a *completed* function execution, f must follow one path from the entry node to the exit node. For the local graph generation, one could run test cases for each function individually to collect traces.

To have more strict policies, we maintain *loop counters* for each graph to restrict the number of repetitions of given flows or flow sequences. For a trace of length l, we look for consecutive repeated subsequence(s) of length $1, ..., \lfloor l/2 \rfloor$. Such a subsequence indicates the function sending a set of requests repeatedly. We treat such subsequences as a single flow, or *grouped flow*, and use a loop counter to count the repetition of the subsequence. We only need to maintain counters for the nodes whose counter is greater than one. An example is shown in Figure 2.3.

Handling user-input via URL replacement. To identify the URLs that may be constructed based on user input, we group all the URLs extracted from the traces associated with a function by their longest common prefix (LCP) with the following restriction: given a set of URLs $U = \{u_1, ..., u_n\}$, two URLs u_i and u_j are grouped only if their longest common prefix $LCP(u_i, u_j)$ is longer than $LCP(u_i, u_k)$ and $LCP(u_j, u_k)$ for any $u_k \in U$ $(k \neq i, k \neq j)$. Then, if the number of unique URLs in a group is more than a threshold t_{lcp}, we replace the URL in a flow with the LCP of the group the URL belongs to, and append a "$*$" to the LCP to distinguish it from regular URLs. We call the resulting URLs the *LCP URLs*. For example, the three URLs *a.com*, *a.com/test/x*, and *a.com/test/y* will produce two LCP URLs (two groups) *a.com* and *a.com/test/**. We consider that the

URLs in the same group are more related with each other. See §2.7.3 for more discussion.

Global graph generation. In the global graph, each node represents a function. We say that if the execution of a function f' depends on the output of another function f but no explicit messages are exchanged between f and f', there is an *indirect flow* from f to f'. One such example is that f uploads a file to AWS S3, which generates a message to trigger f' to process the uploaded file. We assume that a function depends on the function invoked immediately before it, and use the global graph to capture indirect flows.

The global graph is constructed by observing the final action of each of the functions when the application is made to run different test cases. An edge from node v_i to v_j is added if either (i) node v_i calls node v_j or (ii) v_i initiates an indirect flow to v_j via an external service.

Enforcing policy

Policy enforcement for the local graph. The guard checks the current function state against the generated local flow graph. When a function sends a message, the guard proceeds to the successor node (state) of the current node if the flow matches the successor node, and otherwise blocks the flow. If a node is associated with an LCP URL, the prefix of the destination URL in a legitimate flow should match the LCP. When it is at a node representing a grouped flow, the guard expects to see all the subsequent flows from the function match the flows associated with the grouped flow in order. The enforcement of the local graph does not involve the controller.

Policy enforcement for the global graph. We use *tags*, which perform the same role as the labels in the original CFI, to carry function identity information. The tags are carried in the HTTP headers so guards can remove or add tags without changing messages, and services that do

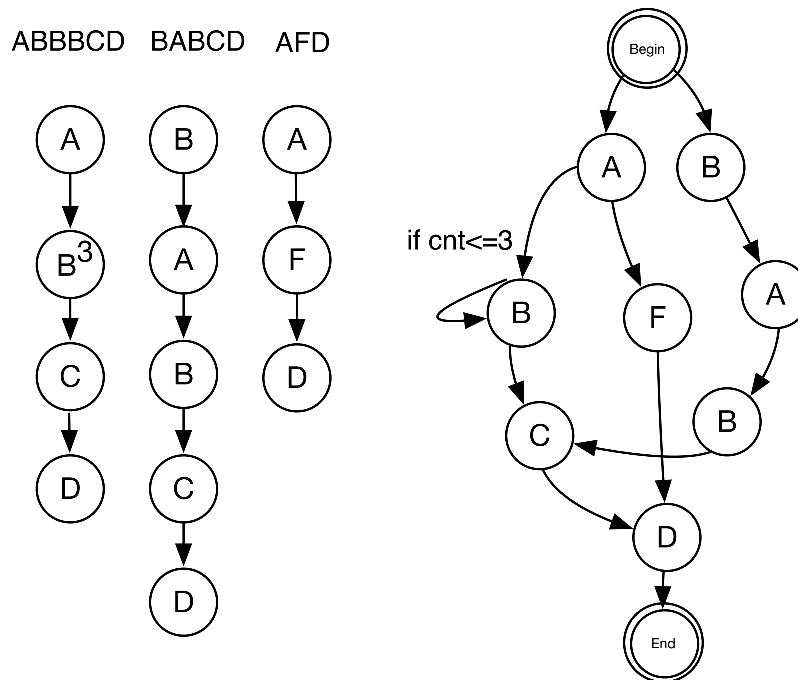

Figure 2.3: An example of the generated local flow graph for three traces ABBBCD, BABCD, and AFD.

not support Kalium will simply ignore this header field and process the messages as normal.

All guards run the following protocol:

(1) If m_{in} comes from another function in the application, the guard extracts the tag $\bar{t}$ from m_{in} and checks the function name in $\bar{t}$ to see if the source of m_{in} is legitimate, as specified in its local flow graph. Otherwise, the guard checks with the controller, which maintains the global flow graph and the global state of the application, to see if its preceding function f' in the global flow graph $(f' \rightarrow s \rightarrow f)$ is at a legitimate state, i.e., whether f' has sent messages to s. Recall that the guard will forward received events asynchronously to the controller so the controller is aware of the

states of all functions (See §2.5). The guard then gets the tag of f' from the controller and saves it as $\bar{t}$.

(2) The guard reuses the request ID in $\bar{t}$ to generate its tag t.

(3) The guard adds t to m_{req}, and removes any tag t' from m_{resp} (t' may not exist if m_{resp} comes from third-party services). The guard will check if t and t' (if it exists) are the same to make sure m_{resp} is sent for it.

(4) The guard adds t to m_{out}. If m_{out} is sent to a function, the request ID in t will be propagated to that function.

Concurrent requests. During trace collection, we only send one request at a time to the application. Likewise, our enforcement mechanism does not support concurrent requests.

2.5.2 Secure runtime

The Kalium secure runtime consists of a modified container runtime runsec which includes a function runtime with instrumented system libraries and an event processing module called the *guard* module. The container runtime runsec provides an API for monitoring and manipulating function execution. The *guard* module uses this API for performing security tasks. runsec also provides a userspace hypercall API for application code to facilitate encryption offloading. Both APIs are shown in Table 2.4. For providing both the runsec API and the userspace hypercall API, the unmodified container runtime must provide a way to view and modify system call events made by the containerized application at a higher semantic level than viewing and modifying the raw system call arguments (as provided by the ptrace system call). To this end, the runtime must provide additional semantic information of the state that is accessed by the system calls.

runsec API. The container runtime, runsec, provides a well-defined API for guards to monitor and control function execution. The `constructEventHook` function creates an event hook for an event specified

`constructEventHook(event_template, syscall_no) -> EventHandle`	Constructs a hook for the system call specified by `syscall_no` and the template which defines the Event type
`waitForEvent(EventHandle) -> Event`	Blocks for an event specified by EventHandle and returns an Event object
`returnDecision(Event, Decision)`	Return the decision for a particular event
`encrypt(cipher, plaintext, key) -> CipherText`	Encrypts plaintext using the algorithm specified by `cipher` and returns the ciphertext

Table 2.4: runsec API and Userspace Hypercall API

by the `event_template` for a particular system call specified by the `syscall_no` and returns an `EventHandle` object. The runtime adds function invocation as a special event that can be hooked. The event template specifies (1) a transformation (parser) from the system call arguments to a serialized string format and (2) the number of system calls whose arguments must be examined for the transformation. Currently, runsec supports event templates for (1) parsing HTTP and TLS traffic on the `SendMsg` and `Write` system calls on sockets, and (2) function invocation. The `waitForEvent` function blocks a guard until an event specified by the `EventHandle` occurs and returns an `Event` object that contains the parameters of the event. The `returnDecision` function specifies the action to be taken by runsec on an `Event`. The actions can include (1) setting the return value of the system call (e.g., return an error to deny the call), (2) allowing the call to proceed, and (3) rewriting the arguments of the system call before running the system call. runsec blocks system calls while the guard processes the event. This generic API enables monitoring a variety of function behavior, although Kalium uses it only for network communication.

Encryption offloading. When functions communicate over an encrypted channel, they pass encrypted data to network system calls that precludes inspection. runsec provides *encryption offloading*, which moves encryption out of the userspace function and into the container runtime, to allow monitoring of the cleartext data prior to encryption. runsec implements a single userspace hypercall API `encrypt`, which encrypts the provided payload using the specified cipher and returns the ciphertext. The runtime caches the ciphertext as a TLS record along with the corresponding plaintext in a queue. Currently, runsec supports the AES-GCM cipher suite widely used in TLS.

When a function sends encrypted data over the network (`SendMsg` or `Write`), runsec interposes on the system call and compares the arguments against the first entry in the cache for a match. If the record matches, the corresponding cached plaintext is used to construct an event. A mismatch indicates that the function either did not use the `encrypt` hypercall or modified the result prior to sending it over the network.

Finally, runsec provides instrumented system libraries to the function runtime that adds calls to `encrypt` as part of the TLS implementation. While we have only implemented encryption offloading for TLS, this architecture can be extended to other protocols such as SSH. Only an application that does not use encryption for messages is allowed to opt out of encryption offloading. If an application bypasses encryption offload or implements its own encryption, the data sent will not match the expected state and the guard will fail the call.

Event processing. The runsec API generates events when functions make system calls, and guards subscribe to these events to monitor and limit behavior. Guard functions execute in response to events to check if the system call is legitimate. To do this, the guard maintains a state table that implements the NFA for the local and global flow graphs. Using the current state and the requested system call, the guard determines the action to be

taken for the operation associated with an event, and returns the decision to the runtime using the `returnDecision` function.

Some guard functions may require knowledge (i.e., global state) from the controller. For instance, as we show in §2.5.1, the guard may need to check whether an event "forwarded" by a third-party service is generated by a legitimate function. Such information can only be obtained by the controller, which has the global view of the application. In this case, the guard will communicate with the controller via an *updater* daemon to check the application global state. To help the controller to reconstruct the global state, the guard forwards any received events to the controller asynchronously via the updater.

2.5.3 Controller

The controller provides a centralized interface for a tenant to manage and distribute customized guard functions, policies, and configurations. The controller distributes updates (local flow graphs) to the different *guards* through the updater daemon during function startup. It also serves as a centralized logger and collects function states from all guards to maintain the global state of the application. The global state is used by the controller to facilitate decision making on function operations which involve indirect flows, such as when storing an object triggers a function to run. For example in Figure 2.1, when the `UpdatePhoto` function sends a message to the user's S3 bucket, the *guard* module in the `UpdatePhoto` function notifies and waits for a decision from the controller on whether that particular message should be blocked or not. The controller tracks the application's state using the global control flow graph and returns a decision based on whether the current state is valid or not.

2.5.4 Extensions to Kalium

Kalium is designed as an extensible, flexible framework that can be used to develop customized guard functions for sophisticated security

tasks. While we currently use the framework only for monitoring network communication, the `runsec` API can be used to monitor and manipulate any system call, which also allows one to emulate an arbitrary protocol's state machine in the guard. Moreover, `runsec` is a container runtime so it can be easily ported to different serverless platforms that are built with different containers. However, in this chapter we focus on the enforcement of control-flow integrity over HTTP and TLS requests.

2.6 Implementation

Secure runtime for serverless functions. The Kalium container runtime runsec is built atop gVisor [152], a lightweight container runtime developed by Google. The gVisor runtime provides an emulation of the Linux kernel over which a containerized application is traced either with a *ptrace* system call or by running the container in a minimal virtual machine using kvm. Briefly, whenever the containerized application makes a system call, it is intercepted by the gVisor runtime which then handles the call as Linux would. GVisor has been integrated into Google Cloud and shows success in mitigating security attacks [151].

Kalium leverages this capability of gVisor to interpose on system calls made by the function to enforce various policies. We modify the gVisor runtime to implement the runsec API.

Inspecting encrypted payloads. We focus on TLS with AES-GCM in our prototype but our method can be applied to other encrypted protocols as well. For HTTPS requests, we restrict the function to use an instrumented version of the OpenSSL library (i.e., `libSSL`) that passes the plaintext data to runsec (before it is encrypted) using the `encrypt` hypercall. Other encryption parameters such as initialization vector and additional data will also be sent along with the plaintext. After returning the ciphertext, runsec encapsulates the ciphertext in a TLS record to get the expected TLS record of the ciphertext, and caches it in a queue. Later runsec will check whether a given TLS record (possibly reconstructed from multiple packets) sent from the function matches the expected record to be sent next.

Guard and controller. The guard is implemented as a module in the gVisor secure runtime. The gVisor routine (i.e., `runsc-sandbox`) that intercepts system calls constructs the event based on the constructed hook (made by calling the `createEventHook` function) and passes it to the guard. The

guard launches the updater in a goroutine during instance startup. The updater serves as the interface between the guard and controller, and communicates with the controller via the updater using *zeromq* [162]. The updater maintains two long-lived `zeromq` connections with the controller, one for sending synchronous events that require the decisions from the controller, and one for sending all events asynchronously to the controller for logging purposes. One can analyze the collected logs to detect abnormal behaviors or to model function control flows (§2.5.1).

The controller is implemented in C/C++ and the other components are implemented in GO, totaling about 1 K lines of C/C++ code, and 2.5 K lines of GO code. We are in the process of developing a full set APIs that can be used for developing guard functions and the management interface.

AWS-based prototype. Many serverless applications are written for AWS and depend heavily on their proprietary services, such as AWS Step Functions. We cannot fully implement Kalium for AWS, as it requires replacing the serverless runtime. To assist in evaluating Kalium on applications that cannot easily be ported other platforms, we implement a version of Kalium for AWS. This prototype launches guards in a separate process and modifies the Python and Node.js modules used by many applications to send events to the guard, and instrument the functions to use the modified modules. We only need to modify the ssl module (`ssl.py`) in Python, and the aws-sdk modules (only `aws-sdk/libs/http/node.js`) in Node.js. The guard is compiled as a binary, and we instrument every function to launch the guard asynchronously in a background process before processing events. This prototype can run in a realistic environment and helps us to more accurately evaluate the accuracy of auto-generated flow graphs.

2.7 Evaluation

We evaluate the security and performance overhead of Kalium. We run our implementation of Kalium in a local testbed running OpenFaas [156] to measure the runtime overhead under various workloads. OpenFaas is a FaaS (Function as a Service) platform that runs on Kubernetes. In addition, we expand the set of complex applications available for testing by evaluating flow graph generation only using the AWS prototype.

2.7.1 Workloads

We evaluate Kalium on three sets of workloads:

Wonderless serverless application dataset. We started with the Wonderless Dataset for Serverless Computing [55], which comprises all open source applications scraped from all public GitHub repositories. As there were *no* OpenFaaS-based applications in the dataset, we port the OpenWhisk-based applications to OpenFaaS. Of the 14 OpenWhisk applications, we evaluate all applications that make network requests: AWS-Text, SMSBot, TwilioTransc and Weather. AWS-Text sends a text message to a number in the request body, SMSBot is a single function application that relays a message to a Slack workspace, TwilioTransc stores a string from the input into an IBM Cloudant database, and Weather returns the weather at a location after querying the OpenWeatherMap API.

Open-source AWS Lambda-based applications. All the applications in the Wonderless dataset comprise a single function, resulting in simple flow graphs. We therefore include other more complicated applications in the evaluation. we study three open-source AWS Lambda-based serverless applications using the AWS-based Kalium prototype: HelloRetail [155], CodePipeline [148], and MapReduce [149]. HelloRetail is a retail platform developed by Nordstrom, and is the most sophisticated open-source serverless application we have seen. CodePipeline is an application from

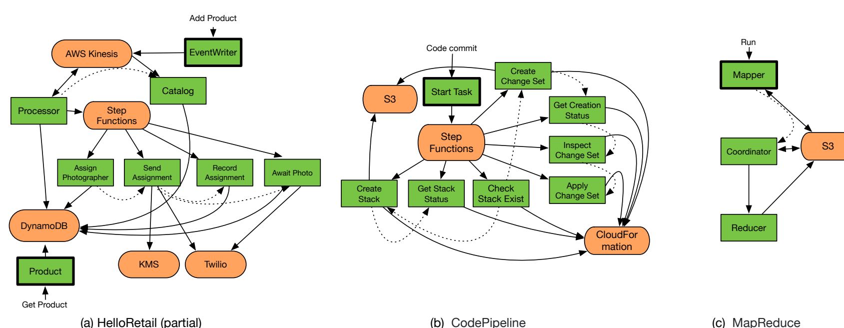

Figure 2.5: Flow graphs of the target applications. Rectangles, rounded rectangles and dotted lines represent function, third-party services, and implicit flows, respectively. The entry functions that accept user requests are highlighted.

AWS for automatically updating the deployment script of software after its source code or configuration has been modified. MapReduce is a serverless-based MapReduce framework. The three applications demonstrate the use of different features of serverless: HelloRetail takes advantage of the event sourcing mechanism and AWS Step Functions to pass messages among functions and invoke function automatically, CodePipeline purely leverages AWS Step Functions to automatically execute functions in order, and MapReduce relies on the auto-scaling feature to run functions in parallel. Table 2.6 and Figure 2.5 show an overview of the three applications and flow graphs.

Valve benchmark suite. Finally, we compare the performance of Kalium with other serverless security frameworks using benchmark suite provided by Valve [47]. The application in the benchmark suite is a reduced version of the HelloRetail (ported to OpenFaaS), that lacks core AWS Step Function and KMS components of the original application.

	LOC	#Func	#Lib	Language
HelloRetail	5,127	12	158	Node.js
CodePipeline	2068	9	112	Node.js
MapReduce	747	3	1	Python

Table 2.6: An overview of the lines of code, number of functions and third-party modules, and the languages of the target applications.

2.7.2 Efficiency of flow tracking

Local testbed setup. We set up a single control plane Kubernetes cluster of five nodes in CloudLab [52], with each node running on a machine of an Intel Xeon E5-2630 2.40GHz CPU and 64 GB RAM. Each machine is connected to a star topology LAN network with a speed of 25 Gbps. The Kalium controller runs on a separate identical node outside the LAN but on the same datacenter. The version of OpenFaas that we used was 8.0.7.

Flow graph generation (training stage). For AWS-Text, SMSBot, TwilioTransc and Weather, we ran the Wfuzz web application fuzzer [161] with the appropriate JSON input templates. We choose fuzzing because the individual functions were simple and had few (< 20) execution paths. Note that fuzzing is just one of many possible ways to generate traces, and we use it strictly for evaluation purposes.

The input parameters in all of these programs are single strings and were generated using a short (< 10 character) permutation of the printable ASCII charset. For HelloRetail, CodePipeline, and MapReduce we started with writing a minimum set of test cases that cover all the execution paths of the target applications. Based on their documentation and state machines (provided by Step Functions), we created 4, 3, and 1 test cases for HelloRetail, CodePipeline, and MapReduce, respectively. Several functions are triggered only when there are errors during request processing. In one round of tests, we run all the test cases once with random user input. In HelloRetail, user requests are for creating products,

to register photographers, and to query products, so we generate random product/photographer information and query strings. CodePipeline will be triggered automatically when there are changes to the monitored repository, and we add 1–5 random files (1 KB) in the monitored repository. For MapReduce, the job is fixed as word count, and user requests specify the dataset and number of files to be processed. We randomly choose one dataset from the *text/tiny/rankings* and *text/tiny/uservisits* datasets in the Big Data Benchmark from AMPLab [158], and process 1–9 files in a request (the datasets only have 9 files). All the other settings remain default.

We obey the limits (request rate, input size, etc.) of all the APIs when constructing the requests, because violating these limits will not generate any new paths, but may cause Step Functions to be stuck for minutes, which unnecessarily prolongs the experiment time.

We ran the test for 1,000 rounds. After analyzing the collected traces, we find that the control flows of all the functions (except for the two that do not make any network requests) differ somewhat from the anticipated flows we get from manual code analysis, even for those simple functions. The reason is uncontrollable factors that are not related to function source code, meaning that they cannot be handled via static code analysis and may introduce unexpected flows. We identify three causes:

(1) *API library implementation*: API libraries may add randomness to endpoints' URLs. For instance, the AWS API for sending data to AWS Codepipeline (used by the `CreateChangeSet` function in CodePipeline) will automatically append a short, random string to the URL of AWS Codepipeline to distinguish different requests.

(2) *Service configuration*: Server-side configuration such as HTTP redirection will introduce extra flows that cannot be inferred from code. To be concrete, a single HTTP present in the function code may translate to two or more requests depending on the configuration of the server.

(3) *Network failures or other unknown service behaviors*: In some cases, the function will retry a request if the request failed due to unstable network conditions or received duplicated messages from services.

Using static analysis may not be able to capture the dynamic behaviors during runtime. For example, if in its code the function only invokes the HTTP GET method once, a static analyzer may assume only one GET request is allowed, while multiple GET request can be sent due to redirection or network failures.

As expected, flows may vary according to user input: In MapReduce, the `Mapper` function will download data from S3 multiple times depending on the size of user data. Similarly, the `Coordinator` function will create different numbers of reducers based on user data. Besides, the exact URLs for the files in S3 will also vary.

The flow graphs of the Wonderless applications comprise a single node representing the least common prefix of the URL of the API endpoint that the function contacts. For the AWS applications, the final local graphs contain only 1–9 nodes (excluding the begin and end nodes). The median size (i.e., number of nodes) of the local graphs for HelloRetail, CodePipeline, and MapReduce are 3, 2, and 7, respectively. The global graphs generated are consistent with the state machines in Step Functions or the expected graph got from manual inspection. Maintaining these graphs in memory incurs negligible memory overhead.

Graph accuracy. Intuitively, more rounds of tests during the training stage will produce more accurate flow graphs. We estimate the false-positive – guards incorrectly block flows and cause function failures – rates of flows graphs over 1,000 rounds of tests, when the flow graphs are built with traces collected from the first n rounds of tests in training. As shown in Figure 2.7, false-positive rates are about 40%, 80%, 65%, 0%, 0%, 0%, 45% and when using traces from only one round, and no false positives when

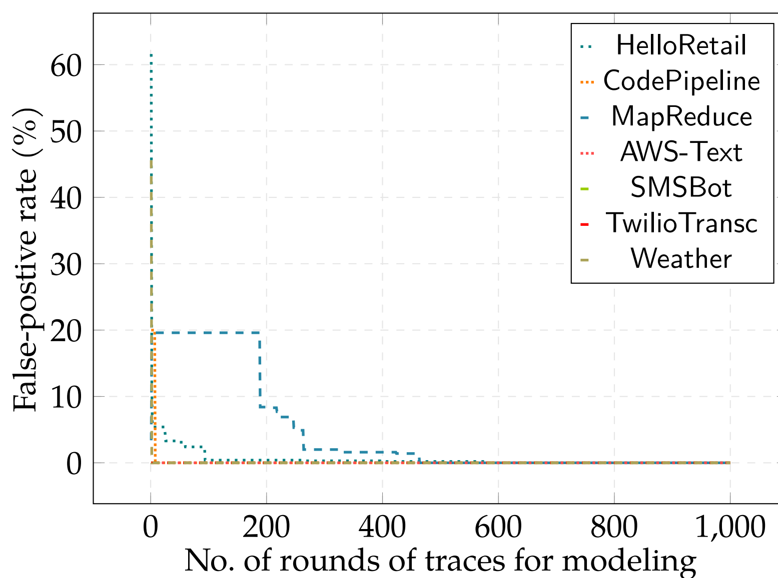

Figure 2.7: False-positive rates when using the traces collected from different numbers of rounds for building flow graphs.

using 583, 7, 463, 1, 1, 1, 2 rounds of traces in HelloRetail, CodePipeline, MapReduce, AWS-Text, SMSBot, TwilioTransc and Weather respectively.

2.7.3 Security analysis

Altering control flows. The flow graphs generated by our flow graph generation represent relatively tight security policies. Even with a slight difference between the expected URL and the event URL, a flow will be considered as invalid and be blocked. The generated flow graphs do not contain cycles or loops, and any given path in a graph represents a legitimate order of flows or function executions. Therefore, the flow tracking function can prevent an adversary from generating arbitrary flows from the compromised function, and reduces the risk of data leakage by restricting the ways for the adversary to externalize data.

Exploiting legitimate function control flows. The adversary may try to exploit legitimate execution paths, e.g., using a compromised function to send requests repeatedly to DoS a server. The loop counter associated with a node indicates how many times the flow can occur in a normal function execution. If it occurs more than the expected times, the flow will be blocked to prevent potential attacks.

The adversary may try to bypass some functions (e.g., authentication) to invoke a target function directly. Though the resulting flows are legitimate, with a global view of the application the guard can still block such attempts: if the requests come from invalid sources or do not carry correct tags, or the preceding function of the target function is not at a correct state, the execution will be blocked. Basically, we do not allow an application execution to begin from the middle of an execution path.

Control flow injection via race condition. The adversary may try to exploit the race condition between function executions to perform a type of flow injection attacks. For example, the adversary leverages some vulnerabilities in Step Functions to execute `CreateChangeSet` just before a normal execution of `CreateChangeSet` begins. In this case, the controller will assume the adversary-launched execution is legitimate, currently we do not support tracking concurrent requests and leave it for future work.

Attacks against runsec. The payload caching mechanism forces the adversary to use the `encrypt` hypercall in order to send an encrypted request. If a function uses a plaintext protocol, then using an encrypted protocol instead will be detected as a violation of the expected request. If a function uses an encrypted protocol but bypasses the offload, runsec will not have cached the expected payload, and will not find a match in the cached expected payloads when the request is sent out. The request, therefore, will be blocked by runsec. Some system call interposition frameworks [66, 194] may be vulnerable to TOCTOU attacks that exploit multithreading. In runsec, since system call arguments are checked after

copying them onto separate buffers which are subsequently used for further processing, such TOCTOU attacks do not occur.

Discussion As we discussed in §2.7.2, static analysis may not be sufficient for understanding the flows generated under realistic workloads. Using dynamic analysis, security depends on the quality of test cases. The flow graphs will be more accurate with test cases that can cover more paths. However, we may still encounter uncovered cases during normal application executions, which could cause execution failures. One possible option is to switch from fail closed to fail open, i.e., the system records suspicious flows rather than block them.

The nodes with the LCP URLs allow the flows whose destination URLs share the same prefix, which may cause security issues. However, without them the resulting graphs could be too restrictive so that they cannot handle change in user input. For example in MapReduce, without using LCP URLs, the flows for fetching different files (that are in the format of "prefix0000",...,"prefix000n") will produce distinct nodes and any future requests that are not for fetching these files will be blocked. We believe manual inspection of flow graphs is a reasonable way to solve this issue. The tenant can adjust the threshold being used to produce the optimal flow graphs and examine whether the nodes with the LCP URLs are appropriate.

The guard cannot prevent attacks that exploit both legitimate function and application control flows, and data-related attacks (e.g., modify the data sent by a compromised function). However, it is feasible to extend our framework and develop more sophisticated guard functions to achieve finer-grained information flow control for serverless applications.

2.7.4 Performance overhead of Kalium

We evaluate performance of Kalium on the local testbed using the benchmark from Valve [47].

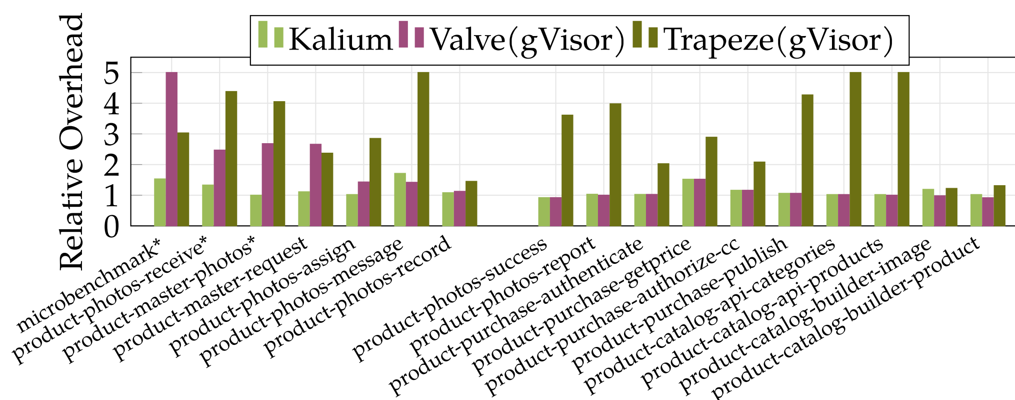

Figure 2.8: The relative latency overhead of the functions in HelloRetail and the microbenchmark function. Valve proxies network requests only in functions marked with an asterisk. Y-axis is truncated at 5.

Performance overhead comparison of Kalium, Valve and Trapeze. We first run the benchmark suite used by Valve [47] to compare performance against prior systems seeking to protect serverless functions. Since our work targets serverless platforms that prioritize security and use gVisor as the default runtime, we assure a fair comparison by replacing the default container runtime of Valve and Trapeze with gVisor.

The benchmark suite provides modified versions of HelloRetail that are integrated with Valve and Trapeze and can run locally. The storage and event queue services used by the originally HelloRetail are emulated with MySQL. For each of 14 functions (16 execution paths) in the modified HelloRetail, we run a `curl` request 100 times with appropriate JSON inputs ($\leqslant$ 1 KB) and record the average request completion time. Additionally, in order to compare the average time taken to run encryption within runsec and the time taken to proxy TLS requests by Valve, we add a *microbenchmark* function which does a GET request to receive an image of size 120K over TLS. The MySQL and image servers are running on Kubernetes nodes in the same datacenter. As the flow graphs for the benchmark are simple, we

also measure the worst-case cost of checking a large graph by checking a single-node graph 1,000 times; we believe 1,000 is a reasonably large size for a local flow graph so the corresponding lookup time represents the upper bound of lookup overhead.

The workloads are:

- product-photos: The workflow involves 8 functions that perform tasks ranging from assigning products to various photographers, to storing the final photographs in the database. The application entry points are the master-request and master-photos functions, which chains 3 functions. The completion times for these two functions are the user-perceived completion times.
- product-purchase: The workflow involves 4 functions that authenticate a user and authorize a credit card payment. None of the functions writes to the database.
- product-catalog: The workflow involves 2 functions that allow products to be added to and queried from the database. Each function has two execution paths.

In Figure 2.8, we show the relative performance overheads of Kalium, Valve and Trapeze with respect to gVisor. The overhead of Kalium consists of the encryption offload overhead and the per-syscall-inspection overhead (§2.5.2). The encryption offload overhead depends on the size of the arguments and the number of times the `encrypt` hypercall is called. The time taken for one `encrypt` hypercall ranges from 314 µs to 434 µs , with an average of 361 µs . For functions that offload encryption, the per-syscall-inspection overhead ranges from 33 µs to 2.4 ms (with global graph query), averaging 1.66 ms. For functions that do not offload encryption, the overhead ranges from 30 µs to 1.81 ms, averaging 606 µs . We can clearly see in most of the cases Kalium has less or similar overhead than

Valve and Trapeze. Recall that Valve uses a MITM proxy to intercept only HTTPS calls. Therefore, Valve only performs proxying or security checks for the functions with an asterisk (i.e., functions that involve HTTP or HTTPS traffic). In contrast, Kalium constructs events for database requests and other protocols (such as DNS and SMTP) for these functions and simulates a dummy policy lookup (similar to HTTP) for each event. Even with the extra security checks, the overhead of Kalium is similar to Valve. The result also suggests MITM-based interception is much less efficient than encryption offloading.

Breakdown of per-syscall-inspection overhead. The worst-case per-syscall-inspection overhead of the order of 2.4 ms occurs when all the critical paths of Kalium are executed. This worst-case overhead includes: (a) parsing TLS records from the raw data passed to the system calls (`SendMsg` and `Write`), (b) TLS record cache lookup (§2.5.2), (c) event construction, (d) guard local graph lookup, and (e) controller global graph query. The breakdown of the overheads is as follows:

- TLS records parsing: The overhead ranges from 6.73 µs to 10.18 µs with an average of 8.16 µs .
- TLS record cache lookup: The overhead ranges from 569 µs to 443 µs with an average of 478 µs .
- Event construction: The overhead ranges from 10 µs to 38 µs with an average of 14 µs .
- Guard local policy lookup: The time of 1,000 lookups ranges from 41 µs to 111 µs with an average of 77 µs .
- Controller global graph query: The overhead ranges from 1.4 ms to 1.9 ms with an average of 1.6 ms.

In the worst-case scenario, the majority of the overhead comes from the communication with the controller, which varies depending on network

condition. We envision that this part of overhead can be eliminated by moving the global graph check inside runsec (§2.8).

2.8 Limitations of Kalium

In this section, we detail the limitations of Kalium and directions for future research.

Policy generation. Our work primarily focuses on using dynamic analysis. A takeaway is that using static analysis or dynamic analysis alone is not sufficient for generating accurate flow graphs (§2.7.2, §2.7.3). One future work is to combine static and dynamic analysis to improve flow graph accuracy. We also plan on exploring automata learning techniques[12] to yield more accurate flow graphs.

Concurrent requests. As mentioned earlier (§2.5.1), Kalium does not handle concurrent requests currently. One way to support concurrent requests is to move the global graph checking inside runsec by propagating the ordered set of tags along each edge in the global control flow graph. In this design, tag propagation must be explicitly supported by external services. An additional benefit of this design is to eliminate the communication overhead introduced by global graph query.

Control flow bending. Control flow bending in regular programs [28] is a technique by which control flow attacks can be mounted even when adhering to the CFG generated in accordance with the strictest CFI policy possible. One possible attack is to manipulate the arguments to functions that are capable of Turing complete computations by themselves, thereby achieving arbitrary computation.

Similarly, in the context of serverless applications, control-flow integrity cannot protect against data integrity attacks in the presence of functions that require no authentication and have unrestricted write access to the datastore. To prevent such attacks either (i) along with the tracking of messages that are passed, the data in each of the messages should also be subject to a dataflow analysis that determines the integrity of data passed

to these powerful functions or (ii) all functions that write to a datastore need to be subjected to authentication.

In Kalium, we have only considered the passing of network messages for the presence of an edge between two functions and do not consider the integrity of the message body.

2.9 Related Work

Information flow control (IFC) can track and restrict information flow in a system, and enforce fine-grained security policies. It has been applied to conventional distributed systems (e.g., DIFC) and cloud applications [20, 139, 140, 164, 211], so, naturally, it can be used for improving serverless security. Trapeze [5] requires modifying all the services being used by the application to support IFC. The requirement of modified infrastructure may make it difficult to apply Trapeze and other similar IFC mechanisms in real serverless applications, which heavily interact with third-party services. Other useful security tools such as event tracing provenance face the same issue in a serverless environment [22, 32, 33, 54, 61, 73, 77, 118, 121, 163, 179]. Compared to Trapeze, Valve provides application transparency, and usability. However, like Trapeze, Valve also requires cooperation from third-party services to propagate taint labels when handling implicit flows through external services. Also, Valve intercepts HTTP(S) requests with a MITM proxy, which may introduce more overhead compared to directly intercepting system calls as in Kalium. Will.IAM [168] introduces network request proxying by explicitly developing applications which route all requests through a proxy. It does not provide any protection in case of a compromised container, where the adversary can bypass the proxy entirely. SCIFFS[143] protects security analytics platforms from information leaks and is complementary to control-flow integrity. Valve, SCIFFS and Will.IAM share common themes of profiling and intercepting system calls as Kalium. Clemmys [187] is a framework that provides confidentiality and integrity of function code and data that is deployed by the users by running functions within SGX enclaves. Similar to Kalium, Clemmys provides protection against the manipulation of the order of function execution. However, Clemmys only supports linear function chains without branches or loops and has a different threat model as

compared to Kalium. In Clemmys, the cloud provider is untrusted, but the function code is trusted and assumed to be bug free.

2.10 Conclusion

Kalium enforces control-flow integrity in serverless applications, both across function instances and within a single function execution. Functions runs in a modified runtime that reports communication events to a guard, which checks whether the communication should be allowed or not based on a function local control flow graph. Complementing the enforcement of local flow policies, a centralized controller collects the states of all functions comprising an application and helps the guard make a decision in the case of implicit flows. We showcase how Kalium can be used to model and monitor application behavior, to prevent flow injection that can lead to data corruption and DoS attacks with minimal overhead.

3 Architecting Trigger Action Platforms for Security, Performance and Functionality

3.1 Introduction

Trigger-action platforms (TAPs) enable end-users to automate interactions between a wide variety of third-party online services and devices [107]. However, it requires that the user place a lot of trust in the operation of the TAP. 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 [2, 76, 129, 191]. 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 [83]. Consequently, some third-party services (e.g., GMail) have become reluctant to interface with TAPs citing privacy concerns [84]. At the same time, users are becoming wary of the insufficient safeguarding of their data by TAPs [50]. In an ideal world, a TAP would only execute user-created applets while ensuring that attackers cannot manipulate them or steal their data.

We contribute to the line of TAP 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 3.9 contains a comparative analysis). Specifically, our work achieves better performance and functionality than eTAP [34] and Walnut [166] while offering similar security (under different assumptions), and stronger security than minTAP [207], DTAP [58] and oTAP [37]. We achieve these desirable points in the design space because of *tailored* use of trusted execution environments (TEEs).

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 [34] or Walnut [166]. 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 3.7.1, 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 [7, 48, 59, 91]. 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 3.4.2). 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).

3.2 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 rule in Figure 3.1. 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 3.8.

For interoperability with third-party services, popular TAPs like IFTTT and Zapier specify a compatibility or shim layer that the participating

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

Figure 3.1: Example applet.

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 [86, 210]. 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 [72, 119, 201].

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 3.4.2 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].

3.3 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.

3.3.1 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 [40, 68] 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 [58, 207], amplifying the risks. A recent line of work studies the security of trigger-action platforms under the assumption that it is untrustworthy [34, 58, 207]. 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 [34, 58, 207].

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 TAPDance is secure and fully trusted.

3.3.2 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.

3.3.3 Functionality Goals

While ensuring the above security goals, we want TAPDance 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.

3.3.4 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 [37]. eTAP also does end-to-end encryption, but can support applet computation using garbled circuits [34]. 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.

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

3.4 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 [11]. 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 [129] 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, 36, 38, 190, 206] 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.

3.4.1 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. As we show in Section 3.7, 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 [30]. 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.[1] Additionally, the trigger and action data types representing inputs and outputs in applets are fully known at compile time, making type inference simple. Table 3.4 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 [34, 58, 207]. 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

[1] 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.

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.

3.4.2 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

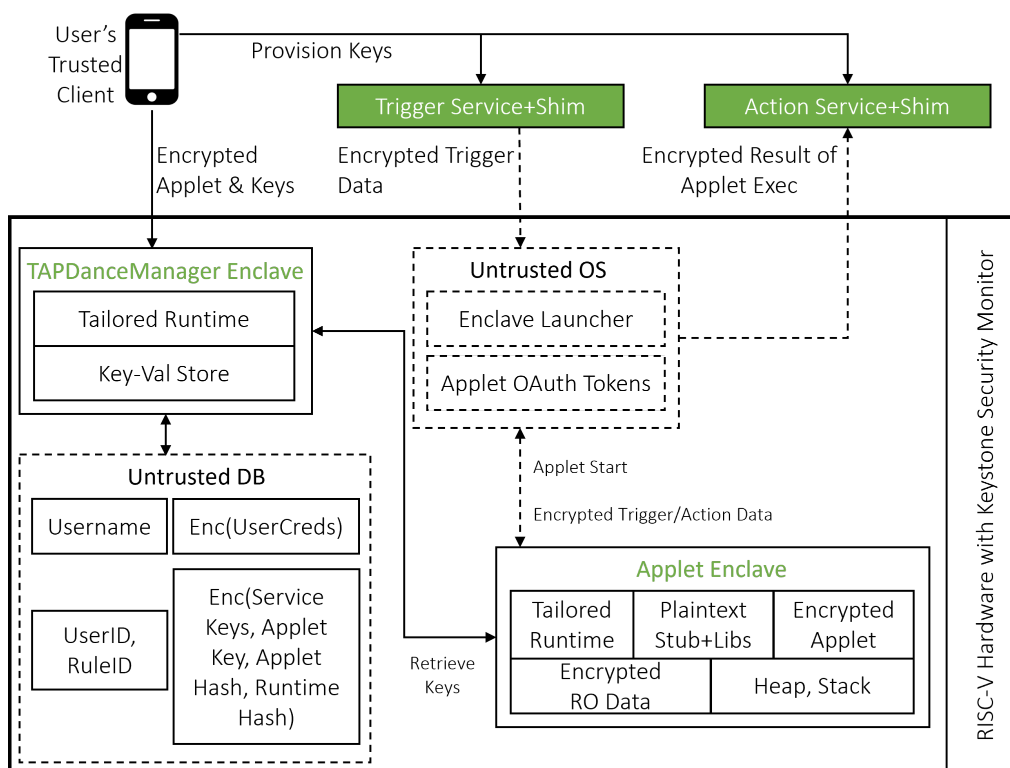

Figure 3.3: The TAPDance Architecture: Based on Keystone enclaves using RISC-V primitives.

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. 3.3 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 3.2. 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 [203], 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 [34, 58, 207]. 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 3.5 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 3.5 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 3.5.3 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 [88], we document our additional restrictions in Table 3.4. Our compiler supports

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

a TypeScript variant similar to AssemblyScript [16], 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 3.7, 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 [51]. 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`.

3.4.3 User Registration and Authentication to TAPDanceManager

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

3.4.4 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

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}

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})

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 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 [192] 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 [180] 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 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.

```
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 3.5: Benchmarking Applet

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.

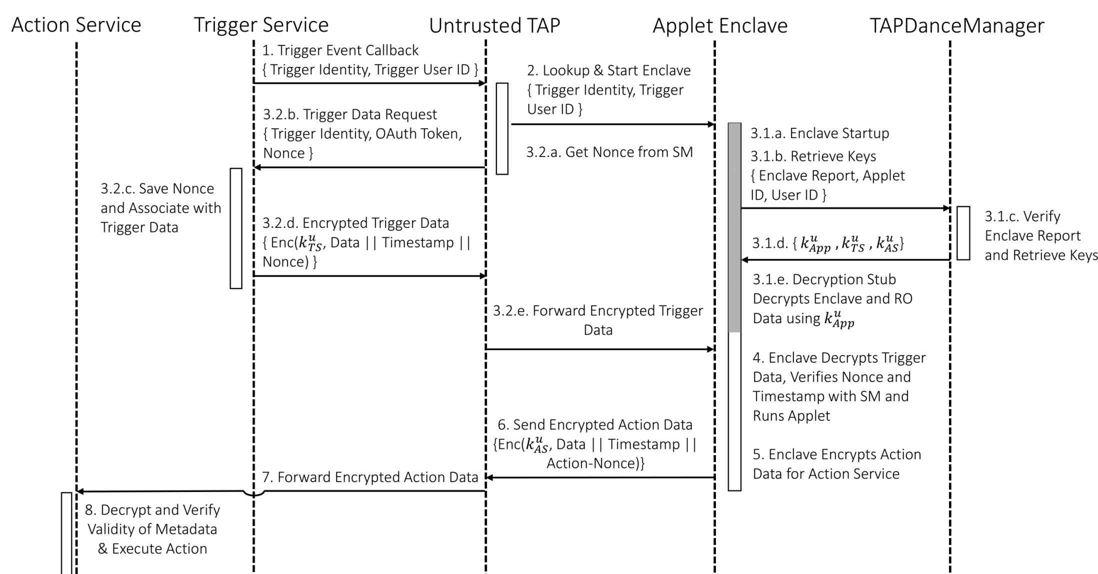

Figure 3.6: The TAPDance Execution Protocol. Sensitive data and results are encrypted everywhere except inside an attested applet enclave and the trusted endpoint services. Steps 3.1 and 3.2 occur in parallel, where the alphabetic identifiers indicate serial steps within those parallel events. The shaded region represents the additional steps that take place on a cold-start.

3.5 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 3.7.

Attested TLS. All communication between trusted parties occurs over attested TLS [74]. 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 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.[2]

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

3.5.1 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 [74]. 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`.

3.5.2 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}.

3.5.3 Running an Applet

Formally, let the trigger service be denoted by T and the action service be denoted by A with the user applet denoted by f. For running the applet f, let the OAuth token for the trigger service be $O_{T,f}$ and that for the action service be $O_{A,f}$. Events are assumed to have a unique identifier id associated with them. Let the API functions that are needed to retrieve the trigger event E and submit the corresponding action $f(E)$ to the action service be $F_{T,f}$ and $F_{A,f}$ respectively. $(E, f(E), id)$ forms a trigger-action tuple, if $f(E)$ is the action corresponding to the event E with unique identifier id.

Let the timestamp at which the trigger event E occurred be τ_E and the timestamp at which the applet enclave receives the trigger event E be $\tau_{E,f}$, and the timestamp at which the action data is generated be $\tau_{f(E),A}$. Let P denote the processing function at the action service which takes as input the action data $f(E)$ and the current timestamp ts to generate an action. It can fail with $\perp$ if it detects an anomaly.

Figure 3.6 shows the applet execution protocol. The main security goals of the protocol are as follows:

- **Confidentiality and Integrity of Trigger and Action Data**: The confidentiality and integrity of the trigger and action data (E and $f(E)$) should be protected from the untrusted TAP and the Internet as a whole.

- **Replay Protection of Trigger and Action Data**: The protocol should maintain the invariant that a single trigger event generated by the trigger service should yield a single action being performed at the action service. That is, the protocol should ensure that the association

between the trigger event E and the corresponding action data $F(E)$ cannot be modified by the untrusted TAP, and the following condition holds:

$$\forall(E, f(E), id)\ \ P(f(E), ts) \neq \perp \implies \forall(ts' > ts)\ P(f(E), ts') = \perp$$

that is, a particular action should take place only once, and if it is replayed, then P flags an anomaly. Note that f can be executed multiple times on the event E to generate multiple copies of the action data $f(E)$.

- **Integrity and Freshness of Trigger Event Processing**: The integrity of the applet execution should be maintained, and the action data generated should be the output of the applet function uploaded by the user, that is, if the applet is replaced by another function f' it should be detected by P. That is, for all valid trigger-action tuples, the following should hold:

$$(valid(E, f(E), id)) \implies \forall(f' \neq f)\ \ P(f'(E), ts) = \perp$$

Trigger events that are delayed beyond a threshold set by the user $\gamma_{E,T}$ should not be processed to yield action data:

$$(\tau_{E,f} - \tau_E > \gamma_{E,T}) \implies f(E) = \perp$$

Similarly, the action service should refuse to process action data that has been delayed beyond a threshold $\gamma_{f(E),A}$.

$$(ts - \tau_{f(E),A} > \gamma_{f(E),A}) \implies P(f(E), ts) = \perp$$

- **OAuth Tokens**: The OAuth tokens provided by the user for the trigger and action services should not be used to execute API calls to

retrieve information not intended by the user, even when the tokens have the privilege to do so. That is, invoking a function other than $F_{T,f}$ at the trigger service should not reveal any data of the user. Similarly, invoking any function other than $F_{A,f}$ should result in P outputting $\perp$.

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 3.6), 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).

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 3.6).

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 3.2, 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.

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_{TS}^{u}. 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_{AS}^{u}. 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_{AS}^{u}. 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.

3.6 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 3.4, 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

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 3.7: Interfaces in TAPDance.

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 3.7 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 3.8: Example Benchmarking Applet

3.7 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 [182]. 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 [178]. 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 [52]. 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 [51]. 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 [199], 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 [186] for parsing JSON data in the applet enclave. The trigger 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 [43].

Dataset. We obtained a dataset of real user applets from the authors of minTAP [207]. 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 3.8 and in the Appendix(Figure 3.5). 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.

3.7.1 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 3.10 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.

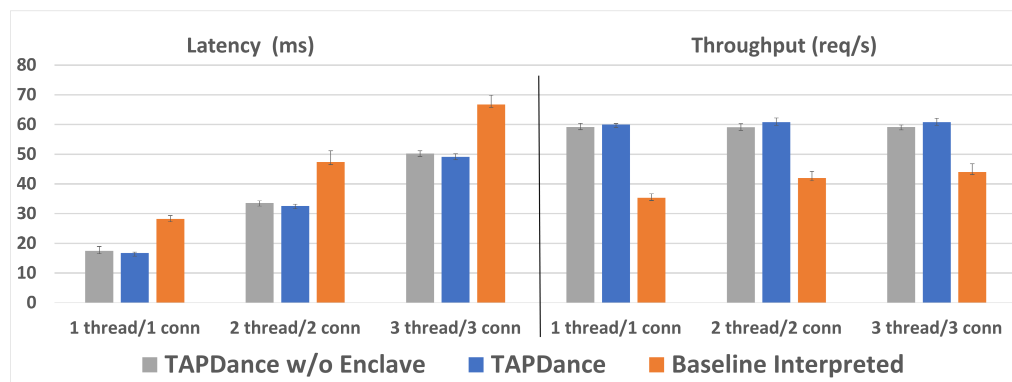

Figure 3.9: Latency and Throughput Comparison of TAPDance with the Baseline TAP that does not use enclaves. TAPDance performs better than the interpreted baseline, and identical to a version without enclaves.

3.7.2 Performance

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

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 3.10: 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 3.11: Performance Metrics of TAPDance.

takes 3s on our prototype board. This reduces to 1ms with hardware hashing support present in modern Intel processors.

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 3.11, 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.

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* [198] 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 3.9 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.

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.

3.8 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 3.4.4.

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 [56, 141].

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 [182].

Criterion	IFTTT	eTAP [34], Walnut [166]	MinTAP [207]	DTAP [58]	OTAP [37]	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 3.12: Comparison of TAPDance with prior TAP security systems. TAPDance offers the best trade-off among all design criteria.

3.9 Related Work

Trigger-Action Platform Security. We contribute to a line of work on re-imagining the security properties of TAPs [34, 37, 58, 104, 129, 197, 207]. Table 3.12 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 [37]. DTAP only supports action integrity without computation on trigger data and does not provide data confidentiality [58]. 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 [34, 58, 207]. 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 [41, 184, 197]. 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 [14], 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 [170] 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 [187] provides a framework for running serverless functions in SGX enclaves. The functions are run inside a SCONE-based environment [171], 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 [24, 65, 146, 187]

Komodo [59] decouples the enclave memory management 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 [78] 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 [176] 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 [36] and Haven [11] 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.

3.10 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.

4 Non-Interactive and Verifier-free Remote Attestation

4.1 Introduction

Trusted execution environments (TEEs) allow clients to run code securely in a harsh environment: machines controlled by an attacker running malicious system software.

Such systems are extremely powerful, but with the TEE enclave itself entirely surrounded by an untrusted operating system, untrusted facilities, and an untrusted network, there is more work to do. To launch an enclave, a client simply sends a piece of code to the cloud provider, which runs it inside a TEE. Since the code passes through the untrusted system, it cannot contain secrets, and before sending secrets, even in encrypted form, a client must verify that (1) the correct code is running, and (2) it is running in a valid enclave that untrusted software cannot pierce, lest an attacker maliciously modify the code or attempt to run it where they can observe its data. Existing TEEs accomplish these goals through *remote attestation*, a protocol that allows a client to verify that the enclave was initialized in the correct state and is running on a genuine TEE-enabled CPU. There are currently two approaches to remote attestation, both with substantial drawbacks.

Traditional attestation, defined by hardware manufacturers like Intel [92, 98] and AMD [9], places the burden of verification fully on the client. The client must receive an attestation from the enclave and verify it with the hardware manufacturer. This model provides very strong security, guarding against an adversarial cloud provider, but clients must run their own verifiers and implement their own secret management, deciding when and how to transmit secrets to a verified enclave. A commercial

grade secret management service such as HashiCorp Vault [75] could be as large as 650K lines of code. Moreover, clients must communicate with each enclave on launch (and to the hardware manufacturer in some attestation schemes), not only requiring them to be online and available, but adding wide-area network latency to the process. This extra latency alone can be prohibitive in applications that require rapid processing of large data, like serverless and high-performance compute workloads. Prior work on protecting serverless functions inside TEEs rely on the client attesting the enclave each time a function is invoked [69, 146, 175] (including the commercial Conclave Cloud Functions [25]). Our measurements show that with a wide-area network latency of 10 ms, attestation increases startup latency 40-400% compared to published AWS cold-start latencies [18].

To solve these scalability and performance problems, commercial deployments, such as those at Microsoft [127] and Amazon [19], place immense trust in the cloud provider. Instead of the client performing their own verification and secret management, the cloud provider does both. Users no longer have to provision secrets to enclaves, and launch avoids wide-area network communication as the cloud provider can place all relevant servers in the same datacenter. However, users must now completely trust the complex services of the cloud provider, reducing the value of TEEs. Intel recently launched a dedicated cloud-based, distributed attestation service called Trust Authority [102] designed to reduce the burden of verification on the users by supporting the offloading of the user's TEE launch policy. In this model, the user still needs to manage secrets and transmit them to the TEE after a successful evaluation of the launch policy by the Trust Authority, and the Trust Authority is a large addition to the user's TCB. Prior work has also used verifiers running inside enclaves [65] running in the same datacenter, removing the infrastructure provider from the TCB. However, managing the state of verifier enclaves

poses the same management challenges as the client running their own verifier, increasing the overall TCB size.

Non-interactive attestation, which does not require online verification, is a promising solution. Previous work on non-interactive remote attestation focused on either deferring the verification of the proof of initial state or relying on periodic verification of the initial state by the remote verifier [108, 120, 173] but does not consider secret provisioning. Furthermore, launch still requires one round trip with the verifier, which is still part of the TCB. Chancel [3] runs a loader enclave that is attested in advance and is entrusted with the secrets needed to decrypt the binaries input by the client. While this approach alleviates the impact of attestation on startup latency, the burden of managing the loader enclaves and provisioning secrets still falls on the client, increasing the size of their TCB.

This work presents Bellerophon, an attestation scheme with the scalability and performance of provider-managed verification and secret management, the traditional threat model trusting the cloud provider only to not deny service, and a smaller trusted computing base (TCB) than prior approaches. The key insight enabling these gains is to encrypt user binaries such that only the TEE inside a genuine and properly-provisioned CPU can correctly decrypt them. As a result, clients can safely include secrets in their initial (encrypted) binary, and be sure *without interactive verification* that only a secure enclave can decrypt them. This assurance eliminates the choice between high trust in the cloud provider and online verification with large network latencies. In addition, it allows binaries to be stored and executed asynchronously, even behind firewalls, at any time at low cost. To maintain compatibility with existing machines capable of running TEEs, Bellerophon does not require any changes to existing cloud machine hardware capable of running TEEs but requires the hardware manufacturer to update their protocols. Each Bellerophon

worker machine requires a persistent and tamper-proof key store that is protected from untrusted software, so we assume a hardware Trusted Platform Module (TPM) in our design.

Designing an encryption-based attestation scheme presents important challenges. First, the successful decryption of the user's binary must depend on the TEE being loaded correctly, just as today's TEE's rely on remote verification to run successfully. This requires great care in the design and, as we discuss in Section 4.4, is not scalable using a standard public-key encryption scheme. Second, as with any protocol sending encrypted secrets through an untrusted channel—in this case a cloud provider—the scheme should be forward secure. That is, a compromise of key material should not leak secrets from prior to the compromise, even if the attacker stored previously encrypted data.

We implement a Bellerophon prototype on Intel SGX enclaves, and show that the Bellerophon's TCB is small: just over 1,000 lines of code for a local architectural enclave for decryption, and 38.6K lines in total including all the other local architectural enclaves, library code and the provisioning server. The addition of a hardware TPM into the TCB is in line with the usage of TPMs for VM attestation in prior academic work and real cloud deployments [128, 133, 188]. Our experiments compare Bellerophon's against SGX-style interactive attestation (also applicable to virtual machine enclaves such as Intel TDX [96] and AMD SeV-SNP [8]) and show that Bellerophon reduces enclave startup latency for a 40 MB TEE binary by 94% when the SGX verifier requires a 10ms network round trip. To demonstrate the effectiveness of Bellerophon in an end-to-end application, we integrate Bellerophon with the reusable enclaves framework [175] to accelerate the startup performance of confidential serverless functions.

The main contributions of the chapter are:

- We describe an attestation scheme based on encryption that is non-interactive, verifier-free, and with the same threat model as that

of interactive attestation. The core insight is the design of a set of architectural enclaves and efficient key management that aid in performing decryption, in contrast to the signing architectural enclaves used in interactive attestation.

- We design a key-management mechanism for TEEs based on HIBE that removes communication with the hardware manufacturer during attestation.
- We design a load balancing scheme for Bellerophon that enables the infrastructure provider to run an encrypted binary on any machine they own. We also design a re-encryption scheme so that the users do not need to re-encrypt binaries for maintaining forward secrecy.
- We show Bellerophon's TCB is smaller and performs better than prior mechanisms by integrating with prior work [175] and running microbenchmarks

4.2 Background

4.2.1 TEEs and Remote Attestation

A Trusted Execution Environment (TEE) is a combination of hardware and firmware that provides both confidentiality and integrity protections even against malicious system software. To ensure a specified computation runs correctly, the TEE prevents system software from accessing the TEE's private data or changing the execution flow of code running in a TEE. These guarantees allow a user to be confident that code will execute as intended and not leak any secrets passed in during computation. The running code and its associated data isolated by a TEE is termed an *enclave*.

For these guarantees to be useful, users need to verify that specified code was properly loaded onto trustworthy hardware. This process, known as *remote attestation*, verifies critical properties of both the enclave state and the hardware. The enclave state includes the entire configuration of its address space (page contents, relative offsets, and permissions), defining precisely what it will do if executed correctly. The hardware state includes the security version number of the CPU, ensuring it properly supports remote attestation and trusted execution, and the version of the associated firmware. The firmware consists of the CPU microcode, and a variety of *architectural enclaves*—software signed by the hardware manufacturer, allowing it to derive the keys needed in the attestation process without external verification, that implements more complex portions of attestation protocols. The full initial hardware and software state is provided by a hardware *measurement*, which returns a hash of all relevant values.

Existing systems perform remote attestation as an interactive protocol. We detail the protocol for Intel SGX [92] here, but other protocols are similar [99].

Provisioning. A TEE user must be able to verify that the hardware running a TEE can be trusted to maintain its security, which means that the user can

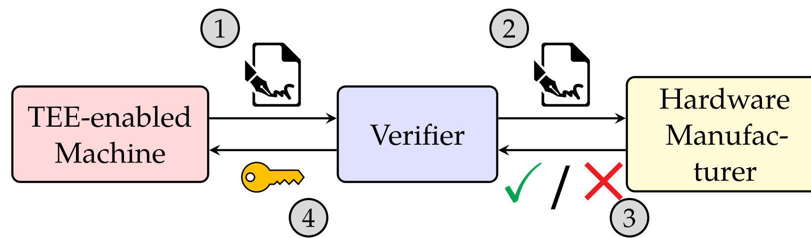

Figure 4.1: Structure of interactive remote attestation: (1) The TEE-enabled machine sends the verifier a quote containing a proof of the initial user TEE state. (2) The verifier forwards the proof to the trusted hardware manufacturer, which (3) responds with the proof's validity. (4) If the proof is valid, the verifier provisions secrets into the TEE.

verify that the hardware comes from a trusted manufacturer. To facilitate this proof, each machine running TEEs is manufactured with an embedded secret called a *provisioning key* shared with its manufacturer. To protect this secret, it is only accessible to a manufacturer-provided *provisioning enclave*. During provisioning, the machine proves its knowledge of the provisioning secret to a provisioning service run by the manufacturer—allowing the service to check the validity of the hardware—and provides its local *firmware version*—ensuring the CPU is not running outdated firmware with known vulnerabilities. For Intel SGX EPID, the secret is a group signing key that can be used to generate signatures over a hash of the initial state of an enclave. To store the secret across reboots, the processor encrypts the secret key with a local sealing key known only to the processor and only accessible with the current firmware version, and stores the encrypted key on local untrusted storage. In Intel ECDSA-based attestation [97], the manufacturer provides the processor with a provisioning certificated signed by the manufacturer and skips the provisioning step. At the end of provisioning, a machine has a signing key tied to a specific manufacturer and firmware version.

Runtime Protocol. When a TEE launches an enclave, it invokes the *quoting enclave*—an architectural enclave with access to the provisioned secret—to

generate a proof of the hardware and initial enclave state that attests to its correctness. In the Intel EPID scheme, the proof consists of a measurement of the user enclave signed with a provisioned signing key.

Figure 4.1 shows how the attestation procedure proceeds using this proof. First, the TEE provides the proof to a *verifier* running on behalf of the user, which then contacts the hardware provider to check its validity. If the proof is valid, the verifier can send secrets (e.g., decryption keys or access tokens) which the user code running in the enclave needs to perform its intended function.

This interactivity is useful for two reasons. First, it enables fresh (ephemeral) secrets on every enclave launch to ensure *forward security*, the guarantee that future compromise of long-term key material (e.g., the provisioned secret) cannot compromise past communications. Second, since the provisioned key is sealed using a key associated with the firmware version, interactive verification can detect rollback to old key and firmware versions. However, it also imposes significant slowdowns, as each hop incurs a wide-area network delay.

Compromise Recovery. If the manufacturer learns of a flaw in the hardware/software TCB of a machine, it can issue TCB updates. Machines that are patched re-run the provisioning protocol to obtain a new group signing key corresponding to the new firmware version. Likewise, the manufacturer can fail verification of any machine known to be compromised.

4.2.2 Hierarchical Identity-Based Encryption

Bellerophon builds on Hierarchical Identity-Based Encryption (HIBE), a public key encryption scheme where each party is assigned a hierarchical identity, any public key is derivable from global public key material and the target party's identity, and private keys are derivable down the hierarchy. More precisely, an identity *id* is a sequence of byte strings $[ID_1, \ldots, ID_n]$

acting as identifiers. Encrypting a message for party *id* requires only *id* and some global public key material. The *master* secret decryption key for the root identity $[]$ is generated randomly, and the $\text{DeriveSKey}()$ API can generate the decryption key for any of its descendants in the identity hierarchy. That is, given decryption key *sk* for identity $[ID_1, \ldots, ID_k]$, $\text{DeriveSKey}(sk, [ID_{k+1}, \ldots, ID_n])$ will produce the decryption key for identity $[ID_1, \ldots, ID_n]$.

This feature enables a hierarchy of key servers. A top-level key server can delegate private key generation for a subset of the ID space. For example, a hardware manufacturer "AMD" could delegate key generation to a cloud provider "AWS" for key identifiers beginning ["AMD", "AWS"]. Clients only need the global public HIBE material (parameters for this specific instantiation of HIBE) and a machine's identity to encrypt a message for that machine.

4.3 Design Considerations

We have three primary goals for Bellerophon: (1) minimize the amount of trusted code required for remote attestation; (2) make enclave launch non-interactive, removing the performance overhead of network communication; and allowing clients to be offline when enclaves launch; (3) retain the strong security guarantees of interactive remote attestation, We first describe the adversarial model for Bellerophon, and then specify our security and functionality goals more precisely.

4.3.1 Adversarial Model and Compromise Recovery

The Bellerophon threat model is largely based on the standard model from the TEE literature [193]. TEE-enabled machines are separated into the self-contained TEE and everything else—including the operating system, network facilities, and storage—referred to as the "untrusted OS." We assume that TEEs, once properly provisioned, will execute correctly and not leak anything—including cryptographic key material—to the untrusted OS not explicitly passed out of the TEE by the software. It is the job of hardware (e.g., SGX, TDX, or SEV) to prevent the untrusted OS from interfering with TEE execution, and Bellerophon does not change the hardware isolation mechanisms. TEEs isolate user code so they cannot access other TEEs or modify the untrusted OS directly. However, the user code can attempt to escape the TEE by utilizing software bugs.

TEE implementations may defend against additional threat models, such as physical attackers that can snoop the memory bus or read/write data in memory beyond the control of the processor. By building on top of existing TEE mechanisms, Bellerophon adopts the threat model of the TEE it builds on. For example, when building on a TEE without memory encryption (e.g., Keystone [48]) the system maybe vulnerable to memory bus snooping, while still defending against an untrusted OS. Notably, we

do assume the hardware TPM on each machine is trusted and tamper-proof.

The Bellerophon system consists of three types of parties.

1. A hardware provider (e.g., Intel or AMD) who manufactures and distributes TEE hardware, provisions private key material, and publishes public keys. This hardware provider is fully trusted.

2. An infrastructure provider (cloud service) who controls the physical machines and their untrusted OSes. The infrastructure provider acts as an active adversary and can deviate from all Bellerophon protocols. In particular, the infrastructure provider is not trusted to trigger key rotations for forward security (Section 4.3.2). Defending against denial of service is beyond the scope of Bellerophon.

3. Clients who request jobs. Clients may attack each other and the infrastructure provider, but will not attack themselves.

We assume that the hardware provider and the infrastructure provider may not collude to compromise user enclaves. Some infrastructure providers manufacture their own TEE hardware such as AWS Nitro and they are outside the scope of this work.

Compromise Recovery. In a practical context, some level of compromise is usually inevitable. We designed Bellerophon and its recovery mechanisms with a simple philosophy: the system should be able to recover from the compromise of any machine that runs arbitrary client code.

To that end, compromise of user code running inside an enclave or provisioned secret keys for an individual cloud machine must be recoverable, with the damage limited only to the compromised job, or jobs running on the compromised machine near in time to when the compromise occurred, respectively. Compromise of burnt-in secrets within

a CPU are unrecoverable for that CPU, but individual CPUs can be revoked, limiting the damage.

We do not consider the ramifications of compromises of any aspect of the hardware provider or of the infrastructure provider's long-term identity keys.

4.3.2 Security and Functionality Goals

We categorise the objectives of Bellerophon into two classes: security goals and functionality (or deployability) goals.

Security goals.

- **S1:** Only enclaves initialized correctly and running on a genuine CPU can access secrets needed for their execution. This prevents the untrusted OS from accessing user secrets.
- **S2:** The user TEE code and data is protected from access by the untrusted system software. This is distinct from S1 that requires protecting secrets.
- **S3:** Enclaves can only access secrets if the hardware and platform software have the specified firmware version. Clients can ensure that TCBs with known vulnerabilities cannot access their secrets.
- **S4:** TEE binaries encrypted in the past are not decryptable by an adversary who compromises a newer decryption key (forward security).
- **S5:** Enclaves can only run in TEEs authorized by the client-specified infrastructure provider and no other.

These security goals correspond to the security goals of current remote attestation mechanisms with interactive verification. However, non-interactivity brings with it a set of unique limitations. First, secrets are

embedded inside the binaries, so secrets are long-lived compared to the ephemeral secrets provided by regular attestation that last only as long as the enclave runs. All the secrets that the enclave has access to are part of the encrypted TEE binary. Second, any policies that decide whether or where an enclave can be launched must be encoded in the encryption mechanism when the TEE is prepared, as there is no longer an interactive service that can enforce policy in real time.

Functionality Goals.

F1: Remote machines can load and run enclaves using only local information, with no communication to services. This reduces the launch latency and reduces the TCB by removing the code to run a highly available verifier service.

F2: The infrastructure provider can run an enclave on a group of machines, not just one. This ensures TEE execution can be a scalable cloud service offering.

F3: The public encryption key for an individual machine must be accessible in a trustworthy way without contacting the hardware provider. This requirement avoids the need for the hardware manufacturer to run a publicly accessible, high volume key server. Ideally, anyone requiring the public key should be able to compute it using public information it read once from the hardware manufacturer.

F4: After a recoverable TCB compromise (see Section 4.3.1) or patchable flaw discovery, it is possible to patch and reprovision a machine, allowing the machine to safely continue operation.

4.4 Bellerophon Design

Bellerophon provides non-interactivity by embedding user secrets in binaries and encrypting those binaries such that only a valid TEE is able to decrypt them. This approach eliminates interactive attestation and secret management, and even allows for storing binaries and executing them later when the client may be offline.

4.4.1 Preparing and Running a Binary

To deploy a computation in a TEE, a user packages their code and secrets into a single self-contained binary *bin* and prepares it for execution using the protocol in Figure 4.2. They encrypt *bin* into an *encrypted blob* using a freshly generated symmetric key κ. They prepend an *encryption stub* to this encrypted blob that, when loaded in a TEE correctly, decrypts the blob and starts executing the TEE binary with its secrets. The user generates a hash H of the encrypted blob and decryption stub and form an *authenticator* by optionally computing additional data ϕ of the same size as H and encrypting $\kappa \parallel H \parallel \phi$ with a public encryption key whose corresponding decryption key is known only to the remote machine.

The additional data ϕ allows the user code to specify the purpose of this decryption request, such as decrypting the initial enclave code or

Binary Preparation

On input $(bin, stub, pk, \phi)$*:*
$\kappa \leftarrow \text{SymKGen}(1^\lambda)$
$blob \leftarrow \text{SymEnc}(\kappa, bin)$
$H \leftarrow \text{Hash}(stub \parallel blob)$
$auth \leftarrow \text{PkEnc}(pk, \kappa \parallel H \parallel \phi)$
output $(stub, blob, auth)$

Figure 4.2: Binary preparation protocol

Decryption Enclave

On input $(auth, \phi)$ *from enclave* $\mathcal{E}$*:*

$sk \leftarrow \text{LoadKey}()$

$\kappa \,\|\, H \,\|\, \phi' \leftarrow \text{PkDec}(sk, auth)$

$H' \leftarrow \text{verifyMeasurement}(\mathcal{E}, H)$

if $H = H'$ **and** $\phi = \phi'$ **output** κ **to** $\mathcal{E}$

else abort

Figure 4.3: Decryption enclave protocol

interpreting a specific encrypted request. By doing so, it supports chained verification. See Section 4.6 for an example.

To ensure the CPU is genuine, the user must obtain the public key for the target machine from the hardware manufacturer in advance. The hash H authenticates the material prepared by the user, and the public key ensures that only a specific, trusted machine can decrypt and execute the code. The decryption stub, encrypted TEE binary with user secrets, and authenticator comprise a *TEE package*.

The TEE platform on a remote machine ensures that only a correct TEE package can execute correctly. Inauthentic hardware or corrupted packages lead to decryption failures, so code and user secrets in the package are inaccessible. The platform provides an architectural *decryption enclave* that has access to the decryption key corresponding to the public key used by the user. As mentioned in Section 4.2, the decryption enclave is architectural and does not have to be attested by the user. The untrusted OS on the remote machine loads the decryption stub and encrypted blob into an enclave and starts the enclave. After measuring the initial memory state of the enclave, the processor executes the decryption stub with the authenticator as an argument. The stub invokes the decryption enclave, passing the authenticator and user-provided data ϕ as arguments. As show in Figure 4.3, the decryption enclave loads its secret decryption key, decrypts the authenticator, and verifies that the caller's measurement and

the user-provided data match the respective values in the authenticator. Matching hashes ($H = H'$) indicates the expected decryption stub and encrypted blob were properly loaded into memory. Matching additional data ($\phi = \phi'$) indicates the running enclave code and the user agree on the purpose of this request. When both match, the decryption enclave returns the symmetric key from the authenticator to the user process, which can then decrypt and run the encrypted blob containing TEE code and secrets. Note that all the communication between the decryption stub and the decryption enclave is encrypted with integrity and replay protection through the use of local attestation [89].

This basic functionality is sufficient to accomplish four of our goals. Both S1 and S2 are satisfied since (a) the authenticator can only be decrypted by a trusted CPU, and (b) the full TEE code and data, including any secrets, remain encrypted until they are correctly loaded into an enclave in an isolated TEE environment. Hence, neither an untrusted machine nor the untrusted OS on a trusted CPU can access user code and secrets. We also ensure S3, because a machine with the wrong firmware version will not have the decryption key for the authenticator. Finally, because a TEE package contains its own authenticator, it can be stored and executed later, even if the client is offline or the machine is disconnected from the network, accomplishing goal F1.

4.4.2 Scalable Key Management

With a classic asymmetric cryptosystem like RSA, the keys for remote machines are unrelated to each other. As a result, users must query the trusted hardware manufacturer every time they wish to run a job on a different machine, violating goal F3. Intel SGX EPID [105] attempts to mitigate this concern by grouping machines under a group signature scheme. All machines in a group share functionally the same signing key, and users only need one verification key per group. This approach scales to large groups, but recovering from the compromise of a single machine

in a group requires re-provisioning *all* machines in the group. These group keys therefore place scalability at odds with resiliency. Certificate-based attestation schemes like Intel ECDSA [9, 97] let infrastructure providers cache public key certificates for specific machines and the user does not need to query the hardware provider for the public key. However, they do not offer forward security - there is no periodic schedule of key rotations except when the firmware version is updated.

Bellerophon avoids this trade-off using Hierarchical Identity-Based Encryption (HIBE), with the hardware manufacturer controlling the root master keys. Each machine receives a unique identity and associated HIBE private decryption key, making compromise recovery simpler and easier. The key material needed to encrypt an authenticator for machine (pk in Figure 4.2) is simply the global HIBE public key material and the identity of the target machine. Public key distribution is also extremely straightforward: the hardware manufacturer makes the single global HIBE public key available. Clients use the same global HIBE public key for every machine, so they only ever need to retrieve it once to derive public keys for any machine. They do have ask the cloud provider for the identity of a target machine.

An identity id consists of a firmware version, the unique hardware identifier for a CPU,[1] and a public signature verification key $vk_{\mathcal{P}}$ identifying the cloud provider $\mathcal{P}$.

Each of these components serves a critical function for accomplishing our security goals. Including the firmware version directly in id accomplishes goal S3: a TEE with an out-of-date TCB will only be provisioned (see below) with the decryption key for some id' with the old version number, so it cannot decrypt authenticators encrypted using an id with the current version number. The CPU hardware identifier ensures that

[1]Intel combines the firmware version and hardware identifier into one value that changes when the TCB updates. Bellerophon can support this behavior, but it limits clients' ability to noninteractively require up-to-date TCBs.

Provisioning machine M **with identity** $id = [cpu, fwver, vk_{\mathcal{P}}]$

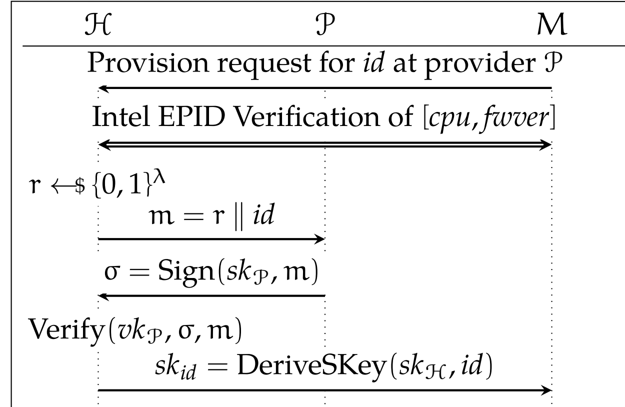

Figure 4.4: The protocol for a hardware provider $\mathcal{H}$ to provision machine M owned by cloud provider $\mathcal{P}$. On a request to provision M with $id = [cpu, fwver, vk_{\mathcal{P}}]$, $\mathcal{H}$ verifies that M has CPU identifer *cpu* and firmware version *tcb* using the existing Intel EPID verification protocol. Before provisioning the key, $\mathcal{P}$ must authorize M through a simple challenge-response. On confirmation, $\mathcal{H}$ derives the HIBE decryption key for *id* using DeriveSKey and provisions it to M.

each physical machine operates with a different decryption key, limiting the damage from compromise and aiding goal F4 (see Section 4.4.3). Finally, as we will see below, including $vk_{\mathcal{P}}$ allows Bellerophon to require $\mathcal{P}$ to sign off on provisioning any machine with this identity. The provider can thus ensure that only machines it controls have identities that include its verification key, supporting goal S5.

Provisioning. Bellerophon's provisioning protocol follows Intel's EPID protocol (see Section 4.2.1) for the hardware manufacturer to verify a genuine TEE-enabled CPU along with its hardware identifier and firmware version. If this verification succeeds, we make two small modifications to the remainder of the protocol, as show in Figure 4.4. First, instead of machines receiving randomly generated keys, they receive the HIBE decryption key associated with their identity, which the hardware manufacturer can derive using the identity and the master decryption key.

Second, to verify that a cloud provider $\mathcal{P}$ authorizes provisioning of any machine with $vk_{\mathcal{P}}$ in its identity, the hardware manufacturer interactively requests a signature from $\mathcal{P}$ as part of the provisioning process. This achieves goal S5.

4.4.3 Forward Security

Forward security ensures that a the compromise of a key at a particular time does not allow an attacker to decrypt objects encrypted at an earlier time [10]. If a single long-term secret were used for a remote machine, compromise of that one key would allow decryption of all binaries ever encrypted for that machine. Bellerophon addresses this problem (S4) by dividing time into *epochs* and rotating keys and destroying old keys every epoch [27]. With key rotation, a key currently in use is unhelpful for decrypting older binaries, greatly limiting the damage from a compromise.

Bellerophon leverages HIBE for two types of key rotations: major and minor. Major epochs define the granularity at which forward security is guaranteed in Bellerophon; a compromised key can, in the worse case, decrypt all binaries encrypted in the same major epoch, but no others. The hardware manufacturer inserts a major epoch counter after the unique hardware identifier for a CPU into the HIBE *id* for a machine. The manufacturer can efficiently generate new decryption keys by changing the major epoch number in an ID, but machines—and attackers in possession of compromised keys—cannot, as they only have a decryption key for the entire ID. The same provisioning protocol described above is used for major epoch rotations.

Each major epoch is further divided into *minor epochs*. Minor epochs further limit the damage that can be caused by an adversary other than the infrastructure provider within a major epoch; a compromise of a remote machine's key at a particular minor epoch does not allow the adversary to decrypt binaries that were encrypted before it, on average cutting in half the window of compromise. We discuss how major and minor

epochs are triggered under the corresponding heading in Section 4.4.3. For *minor epochs*, the new key is derived from the existing key using HIBE by appending an epoch counter to the identifier sequence. This enables every machine to *independently* and without communication update its key and destroy its old key; the prefix property of HIBE ensures that the new decryption key cannot be used to derive the decryption key for an older epoch [27].

Since epochs are time-based, a user can compute major and minor epoch values independently.

The period of major epochs is configurable by the hardware provider and minor epochs by the cloud provider. If these periods are provided to users, they can independently calculate the current epoch from wall-clock time when preparing a binary for execution.

Key rotations present several challenges and opportunities.

Transitions. An entire fleet of machines cannot simultaneously transition from one epoch to the next. In addition, there may be in-flight TEE packages encrypted before an epoch rotation that arrive afterwards. Bellerophon solves this by storing two keys, one for the current epoch and one for the previous epoch, so that binaries that were delayed in transit can still be decrypted. When it stores a new key, Bellerophon permanently deletes an old key.

Efficient representation of epochs As described, each minor epoch lengthens the HIBE identifer sequence. Bellerophon reduces this cost using the HIBE forward security scheme from Canetti et al. [27], that reduces the depth of the HIBE hierarchy to be logarithmic in the total number of minor epochs. Any forward secure HIBE scheme can be used here and we chose this scheme for ease of implementation.

Binary preparation. With epochs, users must include the epoch numbers in the *id* they use for key generation. Users should encrypt binaries with the last minor epoch when the binary can run. As long as that epoch has

not passed, the decryption enclave can derive the associated decryption key from the current epoch's decryption key, allowing the binary to run any time up to the specified epoch.

Re-encrypting binaries. Encrypting with a future epoch number does not work across major epoch boundaries, because the corresponding key is not derivable on the remote machine. As a result, any binary stored for future execution will fail to decrypt after a major epoch rotation.

Bellerophon addresses this with an architectural *reencryption enclave* running on each machine. This enclave has access to the current epoch decryption key and the global public key material. When invoked by the untrusted OS, the reencryption enclave will decrypt the authenticator of a TEE package with the current decryption key and produce a new authenticator encrypted for the next major epoch. The untrusted OS can invoke this service during every major rotation. The system must update all stored binaries before they can run and before the next major epoch, when the old key will be deleted. The Bellerophon authenticator could be extended with a maximum major epoch to limit re-encryptions.

Provisioning: The hardware manufacturer provisions a new machine using the current major epoch and minor epoch of zero. This allows a machine to advance its minor epoch to the current time since the minor epoch can be computed from the current time. Major epochs will increase the load on the hardware manufacturer, as they now must re-provision machines on every major epoch rather than once at machine installation. This cost can be reduced by increasing the length of a major epoch. Alternatively, the HIBE hierarchy enables delegation of major epoch rotations to an intermediate node, such as the cloud provider, at the cost of trusting that intermediary.

Triggering rotations. A correctly behaving infrastructure provider will periodically trigger minor and major epochs. However, a faulty or malicious infrastructure provider is not guaranteed to enforce minor or

major rotations on each machine. A TEE, if it has access to a trusted time source, can perform minor rotations autonomously. An external secure time source could also generate certificates with the current time, allowing the untrusted OS to prove to the TEE that it should rotate its minor epoch forward.

If a provider fails to perform major rotations, the machines will be unable to run new enclaves encrypted with the new major epoch, simply denying service. Users can calculate the major and minor epochs from the current time. Even if a machine does not advance its major or minor epoch, the users will always rotate to the correct epoch and generate new binaries with the keys for the correct epoch (with refreshed secrets every major epoch).

A malicious provider can also *freeze* a machine, taking it offline and stopping key rotations. This attack allows the provider more time to break the decryption key and decrypt any TEE packages with the same major epoch, but is not helpful across major epoch boundaries, as a frozen machine cannot be re-provisioned with the next major epoch key. We discuss this more in Section 4.5.

This forward secrecy mechanism address goal S4, that breaking a current key does not allow decrypting binaries encrypted in the past beyond the current major epoch.

TPM-Based Key Storage. In some TEE systems, like Intel SGX, provisioned secrets are stored encrypted on untrusted storage. This approach poses a serious challenge for forward security. Untrusted software can store copies of old ciphertexts, allowing it to roll back the decryption key to an earlier epoch or and decrypt the old keys later if they compromise a core TEE key. Bellerophon addresses this concern by storing provisioned HIBE decryption keys, encrypted under a hardware-derived long term key, in erasable storage provided by a TPM, which we assume is trustworthy. By

overwriting this TPM memory on key rotation, Bellerophon can ensure that old keys are not captured, even in encrypted form.

Notably, the communication between the provisioning and key rotation enclaves and the TPM passes through the untrusted system software. We therefore employ a standard forward-secret protocol for TPM communication with an active attacker [189].

Compromise recovery. If an attack on the hardware/software TCB of the machine is known, the hardware manufacturer can issue TCB patches. To ensure old firmware versions are not in use (goal S3), the manufacturer and infrastructure provider must re-provision all patched CPUs and publish the new firmware version to users, who use it to encrypt TEE packages. Binaries stored using a key with the old TCB can be deleted or re-encrypted for the new TCB using the re-encryption enclave. For non-critical bugs, this re-provisioning can occur as part of the next major epoch rotation, limiting the expense. For critical bugs, however, it may need to happen "off-cycle," making recovery very expensive.

If the hardware manufacturer learns that a single machine is compromised, for example its provisioning secret leaks, the manufacturer can refuse to issue decryption keys to this machine during provisioning at the next major epoch. This prevents the compromised machine from decrypting TEE packages for future major epochs.

4.4.4 Load-Balancing

As described, Bellerophon requires users to prepare a binary for a specific machine. This is a poor match for a cloud environment, where a binary may be run on any of a cluster of machines, and the set of machines may change over time. We address this with a *load balancer*. This is an architectural enclave that accepts a TEE package encrypted for one machine and the identity of a second, and re-encrypts the authenticator using the public key of the second machine. Like the decryption enclave and re-

encryption enclave, the load balancer is implemented as an enclave, and is authenticated using the same firmware versioning as the other two enclaves. Note that the implementation of the load-balancing enclave is almost identical to the re-encryption enclave. The cloud provider can invoke the load-balancing enclave on any machine satisfying goal F2.

Load balancing is possible within an architectural enclave because HIBE removes the need to know the public key for every machine; the load balancer can use stored global public HIBE key material to generate the public key for re-encryption. To ensure that the target machine is owned by the same provider and is not being redirected to on old, vulnerable machine, the load balancing enclave verifies that only the hardware identifier changes between *id*s, not the firmware version or provider public key.

Under this design, the user encrypts a TEE package for a specific load balancer instance. Bellerophon could support multiple load balancers transparently by issuing them all the same (virtual) machine ID so they share the same decryption key. While this slightly cuts against our compromise recovery goals, load balancers never run user-specified code, reducing the concern.

4.5 Security Analysis

We analyze Bellerophon's security to see how it satisfies the security goals in Section 4.3.2 with a focus on two primary concerns:

1. Security with a malicious infrastructure provider.
2. Security with a malicious user.

Recall that the compromise of burnt-in CPU secrets is outside the scope of Bellerophon.

4.5.1 Malicious Infrastructure Providers

Recall from Section 4.3.1 that Bellerophon assumes the infrastructure provider will not deny service, but otherwise treats it as an active adversary, meaning that it might arbitrarily deviate from the Bellerophon protocols.

S1 and S2 (Confidentiality of User Code and Secrets). Without interactive verification, Bellerophon relies on the confidentiality of user code and data to ensure it only executes on valid TEEs, so the same mechanisms accomplish goals S1 and S2. Recall from Section 4.4.1 that each user workload is encrypted under a fresh secret key κ included in the encrypted authenticator, so these goals reduce to keeping κ secure.

The channel between the user enclave and the decryption enclave is confidential and replay protected by the local attestation Diffie-Hellman protocol [89], so any leak of κ must stem from the user enclave, a leaky load balancer, or leak of the decryption enclave's HIBE decryption key. A correct user enclave is assumed to not leak its own key or secrets. The decryption enclave verifies the initial measurement of the user enclave—the encrypted blob and decryption stub—against the hash provided in the authenticator, thus guaranteeing that κ is only released to correct user enclave code. The load balancer will only ever re-encrypt the authenticator under the identity

of a different machine, so that simply reduces to compromising the HIBE key of the target machine.

The security of the provisioned HIBE encryption key stems directly from the security of the TEE hardware and provisioning protocol. The genuineness of the CPU and the firmware version is verified by the initial steps of the Intel EPID provisioning protocol. The provisioning key a machine uses to prove this can only be derived by a provisioning enclave (i) if it runs on a CPU having a genuine burnt-in root provisioning key and (ii) if the provisioning enclave (with the claimed firmware version) is signed by the hardware provider. The provisioning enclave will not intentionally leak the provisioning key. The Bellerophon provisioning protocol uses the same steps to verify CPU genuineness and firmware version. Therefore the provisioned key accurately reflects the current firmware version.

In Intel EPID, the provisioned key is sealed using a key associated with the firmware version, and interactive verification can detect rollback to previous key and firmware versions. Bellerophon stores the provisioned key in a TPM to avoid downgrade attacks without interactive verification.

A malicious infrastructure provider cannot recover κ without violating one of Bellerophon's security assumptions. It therefore cannot decipher the encrypted blob provided by a user, and is unable to extract any user secrets or the code necessary to improperly run the job.

S3 (Correct Firmware Version). This goal stems directly from including the firmware version in the HIBE identity. The user encrypts their authenticator under an identity that includes a specific firmware version, and the decryption enclave only has access to the provisioned key corresponding to its current firmware version. If those versions do not match, the decryption enclave will be unable to decrypt the user's authenticator, preventing the job from running.

S4 (Forward Security). Forward security is accomplished through the key rotation and the epoch mechanism. As noted in Section 4.4.3, the

infrastructure provider can *freeze* enclaves to prevent them from rotating keys every minor or major epoch. This gives the attacker time to launch attacks on the TEE hardware and firmware guarding the underlying cryptographic keys. If the HIBE decryption key is compromised, the infrastructure provider can decrypt any binaries encrypted for the remainder of the major epoch in place when the freeze occurred. This violates goal S2 for any binaries encrypted for those minor epochs, and goals S1 and S3 for any remaining time in that major epoch. However, such a compromise of a HIBE key provides no benefit across *major* epoch boundaries. By refreshing application secrets each major epoch, users can force the breaking of a new machine every major epoch. Moreover, freezing a machine renders it unable to run user binaries encrypted for future major epochs, reducing the available capacity of the infrastructure provider.

Compromise of a TEE's current provisioning key enables the infrastructure provider to run the provisioning protocol outside the provisioning enclave, leading to compromise of the HIBE decryption keys for all major epochs from the initial point of compromise until that provisioning key is revoked by the hardware manufacturer—which will occur upon detection—or there is a firmware version update. Such a compromise, while damaging, does not violate forward security, since all binaries encrypted for *prior* major epochs remain secure.

S5 (Correct Infrastructure Provider). Similar to goal S3, this security stems from including the infrastructure provider's signature verification key in the HIBE identity, ensuring the binary only runs on hardware provisioned for that provider. Additionally, the hardware manufacturer verifies that key during provisioning (see Section 4.4.2), preventing one infrastructure provider (or a malicious third party) from impersonating another at provisioning.

4.5.2 Fully Malicious Users

A user will not intentionally leak their application secrets. The user can submit malicious code attempting to break free of the enclave and take control of the infrastructure provider's system software. However, this is subsumed by the fact that the infrastructure provider is an active adversary. The user can provide a malformed or invalid authenticator. If the hash of the binary or the symmetric decryption key is incorrect, the enclave will fail to decrypt, leading to a denial of service.

4.6 Case Study: Reusable Serverless Enclaves

Bellerophon is flexible enough to support reusable enclaves [175], a state-of-the-art framework for confidential execution of serverless functions. Reusable enclaves reduce cold starts by running a "function enclave" containing a hardened interpreter in each enclave and reusing it across multiple serverless function executions. The first execution requires standard enclave initialization, but each subsequent invocation simply resets the function enclave to its initial state before executing the user-supplied operation. This optimization reduces a 2–3 second cold start to around 25 ms for state reset, but the existing implementation still requires interactive enclave attestation with users, incurring all problems of interactive attestation including WAN network latencies.

To integrate reusable enclaves with Bellerophon and eliminate the need for interactive attestation, we slightly modify the function enclave to execute the Bellerophon Decryption Enclave protocol and hash user workloads. To prepare a serverless workload, the user encrypts it as described in Section 4.4.1, but uses the hash of the encrypted workload as ϕ and the hash of the correct function enclave as H in the authenticator. On receiving an encrypted workload, the function enclave hashes it, and executes the Bellerophon Decryption Enclave protocol, sending the hash of the workload to the decryption enclave as ϕ. The decryption enclave decrypts the authenticator, verifies H and ϕ match, and sends back the key κ, which the function enclave can use to decrypt and run the workload.

If H in the authenticator matches the hash of the function enclave, then it must be running the correct code, and that code will correctly hash the encrypted workload and send it to the Bellerophon Decryption Enclave as ϕ. Therefore, if H and ϕ *both* match, both the function enclave and the encrypted workload must be unmodified from what the user intended, preventing improper release of κ. This chained verification can extend to multiple layers, with each hashing the previous layer and the decryption

enclave ensuring the chain of hashes produces the expected value. We evaluate the performance of this approach in Section 4.7.

4.7 Performance Evaluation

We evaluate the TCB size and the performance of Bellerophon along four criteria:

1. *Latency of verification:* how long does it take to load and verify a new enclave?
2. *Latency of load balancing:* as load balancing may be done before launch, its cost is on the critical path.
3. *Latency of minor epoch rotation: this is an overhead that reduces the ability to run enclaves.*
4. *End-to-end performance impact of Bellerophon when running serverless functions with the reusable enclaves framework.*

Implementation Notes. We implement a prototype of Bellerophon on top of Intel SGX enclaves. Our prototype prepares user binaries with a decryption stub and symmetric encryption of user code/data. At the remote machine, our prototype implements the full decryption protocol for attestation, the decryption enclave including support for minor epochs, the re-encryption enclave for major epochs, the load balancing enclave, and TPM storage of keys. We implement the provisioning protocol as a server application written in Rust and a provisioning enclave for each machine.

We use Intel SGX SDK v2.23 and the *hohibe* HIBE library [169], a Rust implementation of Boneh et. al.'s HIBE scheme with a constant size ciphertext [45]. We modified the library to run inside an SGX enclave by removing all dependencies on the standard library and replacing them with the enclave runtime components from the Apache Teaclave Project [196]. The system uses a Diffie-Hellman protocol to protect local communication to architectural enclaves [89]. The Bellerophon enclaves

Component	Lines of code
Encryption libraries	17,179
WolfTPM	14,424
Teaclave	1,034
Provisioning Server	1,107
Decryption enclave	1,257
Re-encryption and Load-balancing enclaves (same logic)	1,129
Provisioning enclave	1,555

Table 4.5: Implementation size

access the TPM via the WolfTPM library [200]. We use the protocol described in the CPU to TPM Bus Protection Guide [189] to provide replay protection, confidentiality and integrity of messages sent to the TPM. We encrypt TEE binaries using AES-GCM with the PCL loader encryption tool that is a part of the Intel SGX SDK [101]. The tool inserts a table consisting of the offsets of all the encrypted sections including their respective IVs and tags. The decryption stub uses this table to decrypt all the encrypted sections.

TCB Size. Table 4.5 shows the components of Bellerophon and their size. The enclave logic is dominated by HIBE encryption code. In contrast, Intel's SGX Software TCB including the verifier and the local architectural enclaves is 74,418 lines of code, not including any code for secret management or replication for high availability. Hashicorp's Vault secret manager [75], a reliable replicated service for managing user secrets, is over 450,000 lines of code, similar to what is needed to implement a verifier or secret provisioning service in the cloud.The addition of a hardware TPM into the TCB is in line with the usage of TPMs for VM attestation in prior academic work and real cloud deployments [128, 133, 188]

Testbeds. Our prototype is built on Intel SGX and we test criteria 1-3 (microbenchmarks) on an SGX enabled Intel NUC with an i7-10710U CPU running at 1.6 Ghz with 32 GB of RAM. The NUC is equipped with a TPM v2.0. Note that we are not using server CPUs. The NUC runs Linux 5.10 installed with the Intel SGX PSW and SDK v2.23.

To evaluate the end-to-end impact of Bellerophon on serverless functions, we use an Intel NUC (model NUC7PJYHN1) with an Intel Pentium Silver J5040 processor running at 2 Ghz and with 8 GB of RAM, since it has the Flexible Launch Control (DCAP) feature that is required by the reusable enclaves artifact. The i7-10710U outperforms the Pentium Silver J5040 by 191% on average [185]. This NUC runs Linux 6.8.0-52 with the Intel SGX PSW and SDK v2.23.

To simulate network delays between the user enclave and verifier, we use the `tc` tool to add delay to the loopback network device.

4.7.1 Verification Latency

We separately measure the time taken for enclave creation (load and perform measurement) and to complete the attestation process.

For Intel SGX remote attestation, we run the modified sample code from the Intel SGX SDK which implements the EPID attestation flow [100] with the verifier running as a separate process on the same machine accessed over a TCP/IP socket. This implementation does not count the latency between the verifier and the hardware provider as they are implemented within the same process. To simulate the latency of a system with a low TCB where the users runs their own verifier outside the cloud datacenter, we use the *tc* tool to inject delays into the loopback network interface. The time to create an enclave is the running time of `sgx_create_enclave()` which includes time taken to hash the enclave content - this scales with the enclave size. Interactive remote attestation or the Bellerophon Decryption Protocol is executed after enclave creation.

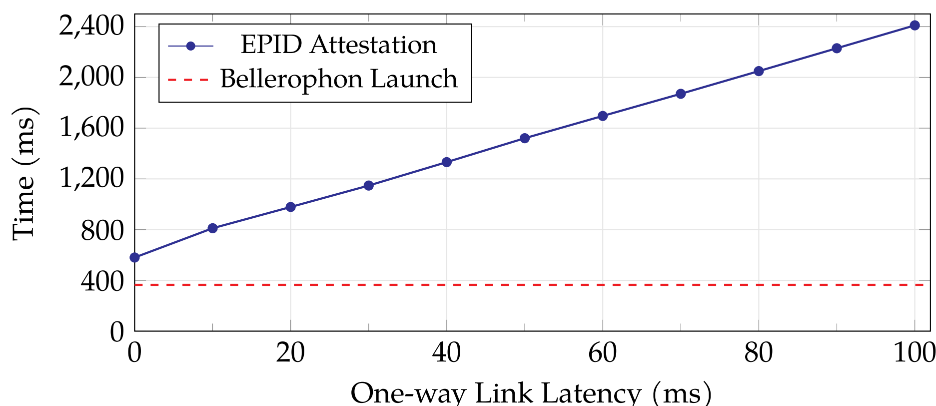

Figure 4.6: Intel EPID remote attestation time increases linearly with link latency. Bellerophon's non-interactive nature means launch does not depend on the link latency. Even for 10k page binaries, Bellerophon's entire launch process (which comprises Intel SGX enclave creation and the Bellerophon Decryption Protocol) is faster than EPID attestation (including the enclave creation time)

Figure 4.6 shows the total latency of completing the interactive attestation process (including the enclave creation time) for various simulated link latencies. All the data points were obtained by computing the average time over 100 runs of the remote attestation protocol. The protocol takes 580 ms (including the enclave creation time) with a latency of 0 ms, and for every 10ms increase in link latency, the protocol latency increases by approximately 160ms. This is because there are 8 messages sent in each direction. The EPID protocol running time is independent of enclave size, as it only sends fixed-size messages. The number of messages is consistent with the Intel Trust Authority prescribed attestation models [99]. We note that the network latency is the dominating factor in the protocol and changing the signature algorithm to ECDSA [97] does not change the overall trend.

For Bellerophon, the decryption protocol replaces the interactive remote attestation protocol of SGX. Recall that Bellerophon does not require a trusted verifier for its decryption protocol, removing the TCB

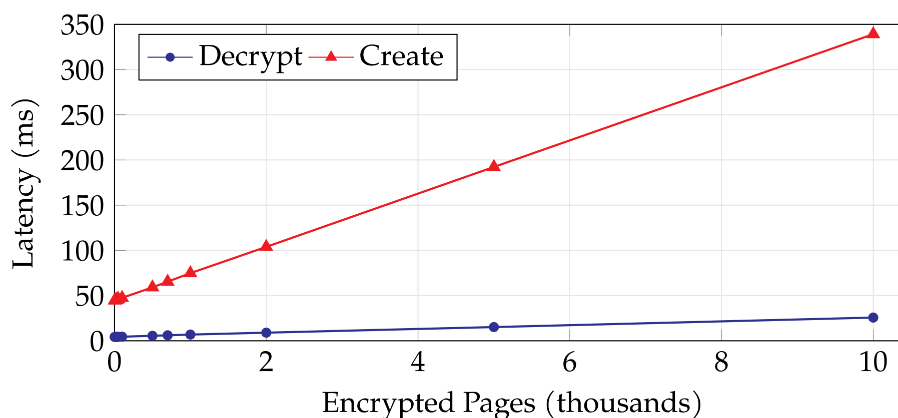

Figure 4.7: Bellerophon Decryption Protocol and Intel SGX enclave creation latency. Note that the Bellerophon Decryption Protocol is executed after Intel SGX enclave creation.

of the interactive verifier. To measure the latency of the Bellerophon decryption protocol, we use a single enclave and vary the number of user data pages in the TEE binary. Since the Bellerophon Decryption Enclave is using a pre-computed HIBE decryption key with [45], the decryption time of the symmetric key blob does not depend on the depth of the HIBE hierarchy. We also measure the time taken for SGX to create the enclave as a comparison. Note that this creation time is the stock SGX enclave creation time dependent on the size of the enclave, and is present for both Bellerophon and SGX attested enclaves.

Figure 4.7 shows the latency of completing the Bellerophon decryption protocol along with the time taken to create the enclave. The decryption of the symmetric key blob takes a constant 4ms while the majority of the decryption time goes in the decryption of the encrypted binary using AES-GCM. An enclave with 10,000 encrypted pages (40 MB) takes 25 ms to decrypt completely. This is 216 ms faster than SGX with zero network latency, and 18× faster than with a link latency of 10 ms. Including the

creation latency, a 40 MB enclaves takes only 365 ms to launch, still faster than SGX EPID excluding the creation time with a link latency of 0 ms.

4.7.2 Re-encryption and Rotation Latency

Bellerophon provides three architectural enclaves. The re-encryption enclave and load-balancing enclave decrypt an authenticator, derive a new HIBE key and then encrypt the authenticator under the new key. In both cases the data size, an authenticator, is fixed, and the performance variability comes from HIBE key derivation; a longer *id* sequence leads to longer key derivations. The decryption enclave performs minor epoch rotations where it derives a key for the next epoch and writes it to the TPM. For all three operations, the majority of the computation is spent in a single DeriveSKey operation. We vary the depth of the HIBE hierarchy (i.e., depth of minor epoch IDs) used for re-encryption and rotation with each identifier being a 64 byte string. The expected depth in depth in practice is not expected to exceed 30; at most 5 for the machine and infrastructure-related identifiers including the major epoch and 25 for the minor epoch. For each depth, we compute the average over 100 runs of the operation. As load balancing is almost identical to re-encryption, we do not report its performance.

Figure 4.8 shows mean re-encryption and minor epoch rotation computation time, which are nearly identical. For short hierarchies, re-encryption takes only 12 ms, and increases by 1–1.5 ms for each level. This time is minor compared to enclave decryption (Figure 4.3). Writing HIBE keys to the TPM takes on average 22 ms. Overall, the computational overhead of all three enclaves is minimal.

The provisioning protocol takes 0.97 seconds with no network latency (0ms) and 1.12 seconds with a network latency of 10ms.

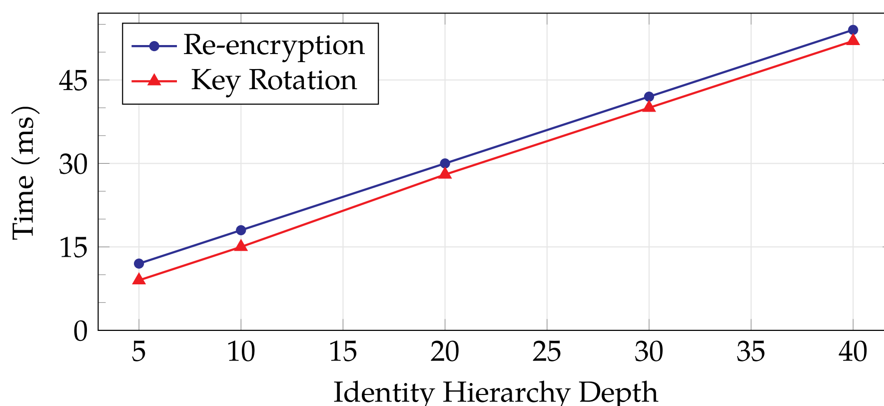

Figure 4.8: Impact of identity hierarchy depth on authenticator reencryption and minor epoch key rotation.

4.7.3 End-to-End Performance

The reusable enclaves framework [175] can be used to run WASM binaries inside an enclave with a WASM interpreter. We integrate Bellerophon into the execution path by encrypting the WASM binaries, generating authenticators as in Section 4.6 and decrypting the binaries by the Bellerophon Decryption Protocol before loading into the WASM interpreter. We measure the time it takes to execute the Bellerophon Decryption Protcol and decrypt the WASM binary before execution. We use the same benchmark functions used by Shixuan Zhao and Pinshen Xu and Guoxing Chen and Mengya Zhang and Yinqian Zhang and Zhiqiang Lin [175]:

- `add`: adds two input numbers
- `prime`: computes the 20,000,000th prime number
- `hash`: hashes a string of length 10, 100 times
- `clock`: measures the granularity of the interpreter's internal clock implementation

Benchmark	Execution Time (ms)	Intel ECDSA (ms)	Bellerophon (ms)
`add`	38	132 + 8×WAN	27
`hash`	42	132 + 8×WAN	27
`prime`	10,716	132 + 8×WAN	27
`clock`	38	132 + 8×WAN	27

Table 4.9: Warm start performance of reusable serverless functions. The times for Intel ECDSA are in terms of the single-hop wide-area network latency, WAN.

Recall from Section 4.6 that the reusable enclaves framework requires that the function enclave be attested per invocation of the serverless function. To compare with Bellerophon, we measure the time it takes to perform ECDSA attestation (that is part of the reusable enclaves implementation) of the function enclave. The sizes of the WASM binaries are in the range of 34-36 4K pages

Table 4.9 shows the mean function execution time in comparison with the time taken for remote attestation as well as for Bellerophon (standard deviation under 5ms for all measurements). For Intel ECDSA attestation, the time is the same for each function since each uses the same function encalve. The measured time of 132ms includes no network delays, but ECDSA attestation incurs 8 wide-area network hops. By contrast, Bellerophon takes only 27ms to run in all of the functions, since the sizes of the WASM binaries are all similar. Bellerophon outperforms ECDSA attestation even with no network delays, and remains independent of the wide area latency. For long running functions, like `prime` which takes nearly 11 seconds, the performance gains of Bellerophon becomes less significant.

4.8 Discussion and Limitations

Applicability to non-SGX TEEs. Bellerophon generalizes to virtual machine-based TEEs such as Intel TDX and AMD SEV since both support securely retrieving the measurement of a user TEE. However, virtual machine launch cost is high, so the added latency of interactive verification may not be significant. TEEs on ARM TrustZone and RISC-V [48, 59, 190] are implemented using a security monitor that controls the lifecycle of the TEE. In both cases, the monitor provides a secure way to get the initial measurement of a user enclave and can run trusted code as part of the monitor, so they can support Bellerophon.

Real-time launch policies and session-based forward secrecy. Due to non-interactivity, Bellerophon does not support dynamic decision making of whether a particular TEE instance should be run or not, such as based on the time of day, as supported by Intel Trust Authority [99]. Static launch policies can be baked into the decryption stub with careful consideration of their dependency on the untrusted system software returning correct data. Dynamic changes to launch policies are not supported and a change to a policy requires the user to generate a new binary and invalidate old secrets. Further, non-interactivity does not support session-based ephemeral secrets in the TEE computation that provide forward *secrecy*.

Applications requiring these features can implement logic in the enclave code to perform interactive checks or request ephemeral secrets. While this structure would eliminate many of the benefits of Bellerophon, it shows that we have not fundamentally reduced what functionality is possible; one can always add interactivity to a non-interactive system.

Enforcing policy in architectural enclaves. The architectural enclaves in Bellerophon could be extended to enforce additional policies. For example, the load balancer described in Section 4.4.4 will re-encrypt any authenticator with a new machine ID. This could be restricted by, for

example, adding an *Allow Load Balance* bit in the authenticator which can be set by users and cleared by a load balancer when re-encrypting. This could prevent the output authenticator from being re-encrypted again for a new machine. Likewise, the authenticator could include a maximum major epoch number to control how many times it can be re-encrypted for a new major epoch, preventing its associated binary from continuing to be runnable—and vulnerable to key compromise—indefinitely.

Some useful policies are enforceable without augmenting hardware enclaves at all. The decryption stub could query the stored epoch and refuse to decrypt and execute the binary unless the epoch was within a particular range. While this does not help guard against key compromise, it does combine with the ability to encrypt under future minor epochs to provide some control over when a TEE binary can execute.

Ownership changes or stolen machines. When a machine is sold, the new owner has to re-provision keys using their public key fingerprint. If a machine is stolen, the adversary can run stored binaries encrypted for the infrastructure provider. However, the hardware manufacturer will not re-provision it at the next major epoch, limiting which TEE packages it can run. Moreover, the machine loses power at least once as it is stolen, the decryption enclave could be augmented to require a signature check similar to that in the provisioning protocol, at start-up to verify its current operator.

Key Compromises. A compromise of a machine's hardware/software TCB, required to compromise the provisioning key or the provisioned decryption key may persist until it is disclosed to the manufacturer and patched. Similar to Intel SGX and other commercial TEEs, it is difficult for the user to ascertain the exact time (major epoch) when the attack vector was discovered, and hence the user must assume that all the binaries encrypted with a lower firmware version have been compromised.

Compromise of the burnt-in root provisioning key within a CPU is unrecoverable for that CPU, but individual CPUs can be revoked by the hardware provider. The hardware provider can keep track of the machines that are unrecoverable using the hardware identifier in the machine identity, and refuse to provision those machines with new private keys during provisioning.

4.9 Related Work

Non-interactive attestation. PodArch [177] proposed an idea almost identical to that described in Section 4.4.1. However, it did not address scalable key management or forward security. Chancel [3] uses an attested program loader that can decrypt encrypted binaries. The user still has to maintain secrets in loader enclaves in different machines depending on where the user enclave is going to run in the future. Bellerophon eliminates the complexity associated with this secret management.

Other works have focused on non-interactive attestation in the context of IoT devices, but there is no notion of a user submitting a binary to an IoT device for outsourcing computation. zRA [173] achieves non-interactivity of attestation in IoT devices where the measurement of the trusted firmware is assumed to be known only to the trusted hardware provider. The device publishes a zero knowledge proof of its current firmware state on a permissionless blockchain, which is then verified against the commitment of the measurement at a later time. SCRAPS [120] achieves scalability of the attestation verifier by doing verification in a smart contract on the blockchain for Pub-Sub Iot networks. Similar work on scalable attestation [4, 6, 15, 21, 29, 53, 82, 111, 112, 113, 114, 147, 183] for Pub-Sub IoT networks provide a subset of the systems and security properties that are provided by SCRAPS. However, a trusted verifier is still required to verify the evidence of attestation generated by the IoT devices, making it unsuitable for achieving the goals that Bellerophon targets.

Remote attestation in commodity hardware. Intel SGX supports attestation using the EPID protocol, which preserves the privacy of the CPU being attested, as well as attestation based on ECDSA certificates. With ECDSA certificates, each CPU has a unique asymmetric key pair called the Provisioning Certification Key (PCK) that is derivable by the Provisioning Certification Enclave (PCE), an architectural enclave (i.e.

signed by Intel). Intel publishes the certificate for each unique PCK public key and the PCK private key is in turn used to sign a fresh attestation key generated by the Quoting Enclave, thereby completing a certificate chain. The certificate chain can be cached by the infrastructure provider with the user verifying the chain. AMD SEV [8], Intel TDX [96] and TPM based systems [70, 126] have a similar scheme for attestation where a certificate chain is established from the hardware manufacturer to a device specific attestation key. ARM has not published a remote attestation protocol for its TrustZone based architectures [122], however previous works [116, 212, 213] have proposed several interactive remote attestation schemes for the same. Similar to ARM, the RISC-V specification does not include a remote attestation scheme. LIRA-V [174] proposes an interactive remote attestation protocol for low-powered RISC-V devices and does not satisfy all the functionality goals of Bellerophon. These protocols still rely on interactive attestation with associated latency and complexity of verification.

Remote attestation in cloud environments. Remote attestation in cloud environments is performed based on infrastructure built on top of the attestation schemes supported by the hardware manufacturers of the CPUs running in the cloud datacenter. At the time of writing, Microsoft provides a unified attestation service for all the TEE-based services [127] that it provides, namely AMD SEV-SNP VMs and containers, Intel SGX enclaves, Intel TDX VMs and TPM based systems. The service follows an interactive protocol with a user-facing secret manager that is in charge of verifying the generated attestation result followed by provisioning the secrets into the TEE. Although the attestation service runs inside a TEE, the source code is available only to government customers. Google Cloud provides similar services [70]. AWS provides a custom hypervisor-based TEE solution called Nitro Enclaves [17]. The root of trust is the hypervisor which provides measurements similar to the registers in a TPM, which are

then endorsed using a signing key along with a certificate chain provided by AWS [19]. All of these solutions require interactivity and a large trusted verifier and hence do not satisfy all the functionality goals of Bellerophon.

4.10 Conclusion

Remote attestation is critical for the security of Trusted Execution Environments. However, existing remote attestation schemes rely on an interactive protocol with a verifier that either the user must manage, requiring long-latency communication paths, or is managed by the cloud provider, which greatly increase the trust required of the provider. Bellerophon provides non-interactive attestation by encrypting binaries in a key known only to secure hardware on a trusted machine. It removes all communication during enclave launch as well as trust in a cloud provided verifier. Our evaluation shows that Bellerophon provides comparable security to existing interactive attestation mechanisms at much lower latency.

5 Conclusion and Future Work

In this chapter, we summarize the findings of the dissertation, including lessons learned, and future directions.

5.1 Summary

This thesis makes the case that secure cloud applications can be built without sacrificing performance or scalability, with the careful co-design of cloud infrastructure services and leveraging the specifics of execution models. To make our case, we design, implement and demonstrate three systems - Kalium, TAPDance and Bellerophon.

Kalium is a control flow integrity framework for serverless applications that assumes that the cloud provider is a trusted entity. Kalium outperforms the state-of-the-art information flow control systems for serverless applications while detecting and preventing several new attacks.

TAPDance is a trigger action platform that provides guaranteed process-at-most-once semantics for incoming user messages even when the cloud provider is fully malicious. We achieve this through careful design using RISC-V Keystone enclaves, resulting in a lower hardware and software TCB than alternate approaches. Performance results indicate that TAPDance outperforms a baseline TAP implementation using Node.js with 32% lower latency and 33% higher throughput on average.

Bellerophon is a remote attestation scheme that eliminates the need for online attestation verification of trusted execution environments (TEEs). Eliminating the verifier results in the user not having to trust or even run the infrastructure running the verifier, resulting in a lower TCB and lower startup latency of TEEs. Moreover, Bellerophon seamlessly integrates with existing approaches to accelerate confidential serverless functions designed to reduce launch overheads. Finally, our evaluation shows that

Bellerophon provides similar security to existing interactive attestation mechanisms with much lower latency.

5.2 Lessons Learned

In this chapter, we discuss some of the core lessons learned while working on the various parts of this dissertation.

Simplify as Much as Possible One of the guiding principles that helped a lot during the design phase of each project was to simplify each aspect as much as possible from the ideation phase onwards. During the design of Bellerophon, we were fortunate enough to articulate the core problem early on, which enabled a simple design that leveraged a significant amount of existing machinery. Every new project has a significant challenge in terms of communication. Visualizing, to the extent possible, why a certain decision would result in the desired outcome and trying our best to articulate that in simple terms helped greatly in communication with collaborators and peers, as well as receiving valuable feedback.

Focus on the Overall Story During the initial stages of the PhD program, it was easy to get lost in the weeds, exploring a very specific technical subproblem within the overall project. Most of the time this resulted in short-sightedness, losing focus on the overall story. In a fast-moving field such as Computer Science, it is very helpful to spend some time everyday thinking about how the project is different from prior and concurrent work to accurately position the project. If a certain subproblem is known to be solvable (such as debugging a crash), then solving that can be deferred to accommodate more important problems (such as doing literature survey and deciding what encryption scheme to use) that would be crucial for the overall story.

Strike a Balance Between Expectation and Reality It seems that it is always possible to perform research that satisfies personal curiosity, satisfies the agency funding the research, and also solves some practical

problem. In this fast-changing field, being flexible and agile with research directions/curiosity seems to be crucial.

Make Working Artifacts Whenever Feasible For each project, making working and reproducible artifacts has always been a very rewarding activity, both in terms of learning new implementation skills as well as convincing reviewers of the practicality of new ideas. Building systems with practical and performant implementations is always crucial for real-world adoption.

5.3 Future Work

Scalable Transparent Execution of Cloud Applications Verifiable computation allows users to get proofs of execution stating that their program ran correctly, along with the result. Existing techniques for achieving this include cryptographic techniques that incur a high prover overhead [60, 110, 138, 167] making it unsuitable for user-facing Internet applications. Trusted Execution Environments generate a hardware-backed proof of the initial state of the loaded program. Although this ensures the correct execution of a single program, it does not generate a holistic proof of the entire application execution that may span multiple programs such as serverless applications. Building systems that generate holistic program proofs for large Internet-facing applications seems challenging and exciting.

Proofs of Execution Attributes In Bellerophon, we show how it can be ensured that a user's binary only runs on a cloud provider-owned machine. Exploring proofs of additional attributes such as the time of execution and the location of the datacenter on which the machine was executed seems to be an interesting problem. In other words, how can we build publicly verifiable proofs of the attributes of remote program execution?

5.4 Closing Words

In this dissertation, we have explored several new avenues to co-design cloud applications with cloud infrastructure to enable applications that satisfy a greater number of security properties. We believe that this will serve as a starting point for further exploration into the design of cloud infrastructure that can provide better security for the end user without sacrificing performance. In a world where large amounts of data are processed in the cloud, providing additional security guarantees has been steadily gaining traction with commercial offerings of secure multi-party computation [205] and fully homomorphic encryption [81]. We believe this thesis complements these efforts in providing better security for cloud applications.

Bibliography

[1] Abadi, Martín and Budiu, Mihai and Erlingsson, Ulfar and Ligatti, Jay. Control-flow integrity. In *CCS '05*. ACM, 2005.

[2] Activision Breach. 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`, 2023.

[3] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca and Byoungyoung Lee. CHANCEL: Efficient Multi-client Isolation Under Adversarial Programs. In *NDSS*, 2021.

[4] Ahmad Ibrahim, Ahmad-Reza Sadeghi and Shaza Zeitouni. SeED: Secure Non-Interactive Attestation for Embedded Devices. In *WiSec*, 2017.

[5] Alpernas, Kalev and Flanagan, Cormac and Fouladi, Sadjad and Ryzhyk, Leonid and Sagiv, Mooly and Schmitz, Thomas and Winstein, Keith. Secure Serverless Computing Using Dynamic Information Flow Control. *arXiv preprint arXiv:1802.08984*, 2018.

[6] Ambrosin, Moreno and Conti, Mauro and Ibrahim, Ahmad and Neven, Gregory and Sadeghi, Ahmad-Reza and Schunter, Matthias. SANA: Secure and Scalable Aggregate Network Attestation. In *CCS*, 2016.

[7] AMD. AMD Secure Encrypted Virtualization (SEV). `https://www.amd.com/en/developer/sev.html`, 2017. Accessed on April 18, 2023.

[8] AMD. AMD SEV. `https://www.amd.com/en/developer/sev.html`, 2021.

[9] AMD. AMD SEV-SNP Remote Attestation. https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf, 2022.

[10] Anderson, Ross. Two remarks on public key cryptology. http://www.cl.cam.ac.uk/ftp/users/rja14/forwardsecure.pdf, 1997.

[11] Andrew Baumann and Marcus Peinado and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI '14*, 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[12] Angluin, Dana. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 1987. URL https://doi.org/10.1016/0890-5401(87)90052-6.

[13] AquaSec. Security for Serverless Functions . https://snyk.io/blog/launching-snyk-for-serverless/, 2017.

[14] Arnar Birgisson and Joe Gibbs Politz and Úlfar Erlingsson and Ankur Taly and Michael Vrable and Mark Lentczner. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *NDSS*, 2014.

[15] Asokan, N. and Brasser, Ferdinand and Ibrahim, Ahmad and Sadeghi, Ahmad-Reza and Schunter, Matthias and Tsudik, Gene and Wachsmann, Christian. SEDA: Scalable Embedded Device Attestation. In *CCS*, 2015.

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

[17] AWS. AWS Nitro Enclaves. https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html, 2020. Accessed April 2025.

[18] AWS. AWS Lambda Cold Start. https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/, 2022.

[19] AWS. AWS Attestation. https://docs.aws.amazon.com/enclaves/latest/user/set-up-attestation.html, 2023.

[20] Bacon, Jean and Eyers, David and Pasquier, Thomas FJ-M and Singh, Jatinder and Papagiannis, Ioannis and Pietzuch, Peter. Information flow control for secure cloud computing. *IEEE Transactions on Network and Service Management*, 2014.

[21] Bampatsikos, Michail and Ntantogian, Christoforos and Xenakis, Christos and Thomopoulos, Stelios C. A. BARRETT BlockchAin Regulated REmote aTTestation. In *WI Companion*, 2019.

[22] Barham, Paul and Donnelly, Austin and Isaacs, Rebecca and Mortier, Richard. Using Magpie for request extraction and workload modelling. In *OSDI '04*, 2004.

[23] Beschastnikh, Ivan and Brun, Yuriy and Schneider, Sigurd and Sloan, Michael and Ernst, Michael D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE '11*, 2011.

[24] Brenner, Stefan and Kapitza, Rüdiger. Trust More, Serverless. In *SYSTOR*, 2019. URL https://doi.org/10.1145/3319647.3325825.

[25] Richard Gendal Brown. Conclave Cloud Whitepaper. https://uploads-ssl.webflow.com/62e0881a72ba0c74c831c6f8/631a090677613438d726bd70_Conclave_Introductory_Whitepaper.pdf, 2021. Accessed April 2025.

[26] Cadosecurity. Denonia: Crypto mining malware. https://tinyurl.com/cadosecurity/, 2022.

[27] Canetti, Ran and Halevi, Shai and Katz, Jonathan. A Forward-Secure Public-Key Encryption Scheme. *J. Cryptol.*, 2007.

[28] Carlini, Nicolas and Barresi, Antonio and Payer, Mathias and Wagner, David and Gross, Thomas R. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security*, 2015.

[29] Carpent, Xavier and ElDefrawy, Karim and Rattanavipanon, Norrathep and Tsudik, Gene. Lightweight Swarm Attestation: A Tale of Two LISA-s. In *CCS*, 2017.

[30] Chandra, Satish and Gordon, Colin S. and Jeannin, Jean-Baptiste and Schlesinger, Cole and Sridharan, Manu and Tip, Frank and Choi, Youngil. Type Inference for Static Compilation of JavaScript. *SIGPLAN Not.*, 2016. URL `https://doi.org/10.1145/3022671.2984017`.

[31] Checkoway, Stephen and Shacham, Hovav. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *ASPLOS*, ASPLOS, 2013. URL `https://doi.org/10.1145/2451116.2451145`.

[32] Chen, Mike Y and Kiciman, Emre and Fratkin, Eugene and Fox, Armando and Brewer, Eric. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[33] Chen, Xu and Zhang, Ming and Mao, Zhuoqing Morley and Bahl, Paramvir. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In *OSDI*, 2008.

[34] Chen, Yunang and Chowdhury, Amrita Roy and Wang, Ruizhe and Sabelfeld, Andrei and Chatterjee, Rahul and Fernandes, Earlence. Data Privacy in Trigger-Action Systems. In *IEEE S&P*, 2021.

[35] Cheriton, David. The V distributed system. *Commun. ACM*, 1988. URL https://doi.org/10.1145/42392.42400.

[36] Chia-che Tsai and Donald E. Porter and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC*, 2017. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[37] Yu-Hsi Chiang, Hsu-Chun Hsiao, Chia-Mu Yu, and Tiffany Hyun-Jin Kim. On the Privacy Risks of Compromised Trigger-Action Platforms. In *ESORICS*, 2020.

[38] Christian Priebe and Divya Muthukumaran and Joshua Lind and Huanzhou Zhu and Shujie Cui and Vasily A. Sartakov and Peter Pietzuch. SGX-LKL: Securing the Host OS Interface for Trusted Execution, 2020. URL https://arxiv.org/abs/1908.11143.

[39] Cilium. Cilium: Container Observability using eBPF. https://cilium.io/, 2019.

[40] Circle CI. CircleCI Breach 2023. https://circleci.com/blog/january-4-2023-security-alert/, 2023.

[41] Cobb, Camille and Surbatovich, Milijana and Kawakami, Anna and Sharif, Mahmood and Bauer, Lujo and Das, Anupam and Jia, Limin. How Risky Are Real Users' IFTTT Applets? In *SOUPS*, 2020.

[42] Confidential Computing. Confidential Computing. https://en.wikipedia.org/wiki/Confidential_computing, 2025. Accessed on June 23rd, 2025.

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

[44] Crust. Crust Overview. https://wiki.crust.network/docs/en/crustOverview, 2022.

[45] Dan Boneh and Xavier Boyen and Eu-Jin Goh . Hierarchical Identity Based Encryption with Constant Size Ciphertext. In *EUROCRYPT*, 2005.

[46] Daniel Elegberun. Netflix System Design-Backend Architecture, 2021. URL https://dev.to/gbengelebs/netflix-system-design-backend-architecture-10i3.

[47] Datta, Pubali and Kumar, Prabuddha and Morris, Tristan and Grace, Michael and Rahmati, Amir and Bates, Adam. Valve: Securing Function Workflows on Serverless Computing Platforms. In *WWW*, 2020.

[48] Dayeol Lee and David Kohlbrenner and Shweta Shinde and Krste Asanovic and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *EuroSys*, 2020.

[49] Denning, Dorothy E. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2), 1987.

[50] 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.

[51] Dmitry Patsura. StaticScript TypeScript Compiler. https://github.com/ovr/StaticScript, 2020.

[52] Duplyakin, Dmitry and Ricci, Robert and Maricq, Aleksander and Wong, Gary and Duerig, Jonathon and Eide, Eric and Stoller, Leigh and Hibler, Mike and Johnson, David and Webb, Kirk and others. The design and operation of CloudLab. In *USENIX ATC '19*, 2019.

[53] Dushku, Edlira and Rabbani, Md Masoom and Conti, Mauro and Mancini, Luigi V. and Ranise, Silvio. SARA: Secure Asynchronous Remote Attestation for IoT Systems. *TIFS*, 2020.

[54] Escriva, Robert and Dubey, Ayush and Wong, Bernard and Sirer, Emin Gün. Kronos: The design and implementation of an event ordering service. In *EuroSys*, 2014.

[55] Eskandani, Nafise and Salvaneschi, Guido. The Wonderless Dataset for Serverless Computing. In *MSR '21*, 2021. doi: {10.1109/MSR52588.2021.00075}.

[56] Evan Johnson and David Thien and Yousef Alhessi and Shravan Narayan and Fraser Brown and Sorin Lerner and Tyler McMullen and Stefan Savage and Deian Stefan. Доверя́й, но проверя́й: SFI safety for native-compiled Wasm. In *NDSS*, 2021.

[57] Feng, Henry Hanping and Kolesnikov, Oleg M and Fogla, Prahlad and Lee, Wenke and Gong, Weibo. Anomaly detection using call stack information. In *IEEE S&P*. IEEE, 2003.

[58] Fernandes, Earlence and Rahmati, Amir and Jung, Jaeyeon and Prakash, Atul. Decentralized action integrity for trigger-action IoT platforms. In *NDSS*, 2018.

[59] Ferraiuolo, Andrew and Baumann, Andrew and Hawblitzel, Chris and Parno, Bryan. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *SOSP*, 2017.

[60] Fiore, Dario and Gennaro, Rosario. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *CCS*, CCS, 2012. URL `https://doi.org/10.1145/2382196.2382250`.

[61] Fonseca, Rodrigo and Porter, George and Katz, Randy H and Shenker, Scott and Stoica, Ion. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[62] Forrest, Stephanie and Hofmeyr, Steven A and Somayaji, Anil and Longstaff, Thomas A. A sense of self for unix processes. In *IEEE S&P*, 1996.

[63] Fortanix. Preventing Money Laundering in a Confidential Computing Way. `https://www.fortanix.com/blog/preventing-money-laundering-in-a-confidential-computing-way`, 2023.

[64] Fouladi, Sadjad and Wahby, Riad S and Shacklett, Brennan and Balasubramaniam, Karthikeyan and Zeng, William and Bhalerao, Rahul and Sivaraman, Anirudh and Porter, George and Winstein, Keith. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *NSDI*, 2017.

[65] Fritz Alder and N. Asokan and Arseny Kurnikov and Andrew Paverd and Michael Steiner. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. In *CCSW*, 2019.

[66] Garfinkel, Tal. Traps and Pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, 2003.

[67] Giffin, Jonathon T and Jha, Somesh and Miller, Barton P. Detecting Manipulated Remote Call Streams. In *USENIX Security*, 2002.

[68] GitHub Breach. GitHub Breach 2022. `https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/`, 2022.

[69] Goltzsche, David and Nieke, Manuel and Knauth, Thomas and Kapitza, Rüdiger. AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting. In *Middleware*, 2019.

[70] Google. Confidential VM Attestation. https://learn.microsoft.com/en-us/azure/attestation/overview, 2024.

[71] Gopalakrishna, Rajeev and Spafford, Eugene H and Vitek, Jan. Efficient intrusion detection using automaton inlining. In *IEEE S&P*, 2005.

[72] Gunawi, Haryadi S. and Hao, Mingzhe and Leesatapornwongsa, Tanakorn and Patana-anake, Tiratat and Do, Thanh and Adityatama, Jeffry and Eliazar, Kurnia J. and Laksono, Agung and Lukman, Jeffrey F. and Martin, Vincentius and Satria, Anang D. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *SoCC*, 2014. URL https://doi.org/10.1145/2670979.2670986.

[73] Han, Xueyuan and Pasquier, Thomas and Seltzer, Margo. Provenance-based Intrusion Detection: Opportunities and Challenges. *arXiv preprint arXiv:1806.00934*, 2018.

[74] Hannes Tschofenig and Thomas Fossati and Paul Howard and Ionuț Mihalcea and Yogesh Deshpande. Using Attestation in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). Internet-Draft draft-fossati-tls-attestation-02, Internet Engineering Task Force, 2022. URL https://datatracker.ietf.org/doc/draft-fossati-tls-attestation/02/. Work in Progress.

[75] HashiCorp. hashicorp/vault: A tool for secrets management, encryption as a service, and privileged access management. https://github.com/hashicorp/vault, 2015. Accessed April 2025.

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

[77] Hossain, Md Nahid and Wang, Junao and Weisse, Ofir and Sekar, R and Genkin, Daniel and He, Boyuan and Stoller, Scott D and Fang, Gan and Piessens, Frank and Downing, Evan and others. Dependence-preserving data compaction for scalable forensic analysis. In *USENIX Security*, 2018.

[78] Hunt, Tyler and Zhu, Zhiting and Xu, Yuanzhong and Peter, Simon and Witchel, Emmett. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. *ACM Trans. Comput. Syst.*, 2018. URL `https://doi.org/10.1145/3231594`.

[79] IBM. IBM VM 50th anniversary, 2022. URL `https://www.vm.ibm.com/history/50th/index.html`.

[80] IBM. Cost of a Data Breach Report, 2024. URL `https://www.ibm.com/reports/data-breach`. Accessed June 2025.

[81] IBM. IBM Z Content Solutions. `https://www.ibm.com/support/z-content-solutions/fully-homomorphic-encryption/`, 2025. Accessed July 2025.

[82] Ibrahim, Ahmad and Sadeghi, Ahmad-Reza and Tsudik, Gene and Zeitouni, Shaza. DARPA: Device Attestation Resilient to Physical Attacks. In *WISEC*, 2016.

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

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

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

[86] IFTTT. IFTTT: Service API requirements. `https://platform.ifttt.com/docs/api_reference`, 2020.

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

[88] IFTTT. IFTTT Documentation. https://ifttt.com/docs/api_reference, 2022.

[89] Intel. Innovative Technology for CPU Based Attestation and Sealing. https://www.intel.com/content/www/us/en/developer/articles/technical/innovative-technology-for-cpu-based-attestation-and-sealing.html, 2013.

[90] Intel. Intel SGX. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2015.

[91] Intel. Intel Software Guard Extensions (Intel SGX). https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2015. Accessed on April 18, 2023.

[92] Intel. Intel SGX Remote Attestation. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html, 2018. Accessed April 2025.

[93] Intel. Intel SGX Helps UCSF Propel Medical Device Innovations. https://www.intel.com/content/www/us/en/newsroom/news/ucsf-propel-medical-device-innovations.html#gs.i6buxf, 2020.

[94] Intel. Maximum Security at the Processor Level: Intel SGX Protects Electronic Patient Record. https://www.intel.com/content/www/us/en/content-details/826053/maximum-security-at-the-processor-level-intel-sgx-protects-electronic-patient-record.html, 2020.

[95] Intel. Intel and Penn Medicine Announce Results of Largest Medical Federated Learning Study. https://www.intc.com/news-events/

press-releases/detail/1593/intel-and-penn-medicine-announce-results-of-largest-medical, 2020.

[96] Intel. Intel TDX White Paper. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html, 2021.

[97] Intel. Intel SGX Data Center Attestation Primitives (Intel SGX DCAP). https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/DCAP_ECDSA_Orientation.pdf, 2022.

[98] Intel. Intel TDX White Paper. https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf, 2022. Accessed April 2025.

[99] Intel. Intel Trust Authority. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html, 2023.

[100] Intel. Code Sample: Intel Software Guard Extensions Remote Attestation End-to-End Example. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/get-started.html, 2023.

[101] Intel. Intel SGX Protected Code Loader. https://github.com/intel/linux-sgx-pcl, 2023.

[102] Intel. Intel Trust Authority. https://docs.trustauthority.intel.com/main/articles/introduction.html, 2023.

[103] Istio. Istio Service Mesh for Kubernetes. https://istio.io/, 2019.

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

[105] J. Li and E. Brickell. Enhanced Privacy ID from Bilinear Pairing for Hardware Authentication and Attestation. In *SocialCom/PASSAT*, 2010.

[106] J. T. Giffin, S. Jha, and B. P. Miller. Efficient Context-Sensitive Intrusion Detection. In *NDSS*, 2004.

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

[108] Jens Neureither and Alexandra Dmitrienko and David Koisser and Ferdinand Brasser and Ahmad-Reza Sadeghi. LegIoT: Ledgered Trust Management Platform for IoT. In *ESORICS*, 2020.

[109] Jonas, Eric and Pu, Qifan and Venkataraman, Shivaram and Stoica, Ion and Recht, Benjamin. Occupy the cloud: Distributed computing for the 99%. In *SoCC*, 2017.

[110] Kasra Abbaszadeh and Christodoulos Pappas and Jonathan Katz and Dimitrios Papadopoulos. Zero-Knowledge Proofs of Training for Deep Neural Networks. Cryptology ePrint Archive, Paper 2024/162, 2024. URL `https://eprint.iacr.org/2024/162`.

[111] Kohnhauser, Florian and Buscher, Niklas and Gabmeyer, Sebastian and Katzenbeisser, Stefan. SCAPI: A Scalable Attestation Protocol to Detect Software and Physical Attacks. In *WiSec*, 2017.

[112] Kohnhauser, Florian and Buscher, Niklas and Gabmeyer, Sebastian and Katzenbeisser, Stefan. A Practical Attestation Protocol for Autonomous Embedded Systems. In *EuroS&P*, 2019.

[113] Kohnhäuser, Florian and Büscher, Niklas and Katzenbeisser, Stefan. SALAD: Secure and Lightweight Attestation of Highly Dynamic and Disruptive Networks. In *ASIACCS*, 2018.

[114] Kuang, Boyu and Fu, Anmin and Yu, Shui and Yang, Guomin and Su, Mang and Zhang, Yuqing. ESDRA: An Efficient and Secure Distributed Remote Attestation Scheme for IoT Swarms. *IEEE Internet of Things Journal*, 2019.

[115] Lazowska, Edward D. and Levy, Henry M. and Almes, Guy T. and Fischer, Michael J. and Fowler, Robert J. and Vestal, Stephen C. The architecture of the Eden system. In *SOSP*, 1981. URL `https://doi.org/10.1145/800216.806603`.

[116] Li, Wenhao and Li, Haibo and Chen, Haibo and Xia, Yubin. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *MobiSys*, 2015.

[117] Ling, Zhen and Luo, Junzhou and Zhang, Yang and Ming Yang and Fu, Xinwen and Yu, Wei. A novel network delay based side-channel attack: Modeling and defense. In *IEEE INFOCOM*, 2012. doi: {10.1109/INFCOM.2012.6195628}.

[118] Liu, Yushan and Zhang, Mu and Li, Ding and Jee, Kangkook and Li, Zhichun and Wu, Zhenyu and Rhee, Junghwan and Mittal, Prateek. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[119] log4shell. CVE-2021-44228 Detail (log4shell). `https://nvd.nist.gov/vuln/detail/CVE-2021-44228`, 2021.

[120] Lukas Petzi and Ala Eddine Ben Yahya and Alexandra Dmitrienko and Gene Tsudik and Thomas Prantl and Samuel Kounev. SCRAPS:

Scalable Collective Remote Attestation for Pub-Sub IoT Networks with Untrusted Proxy Verifier. In *USENIX Security*, 2022.

[121] Mace, Jonathan and Roelke, Ryan and Fonseca, Rodrigo. Pivot tracing: Dynamic causal monitoring for distributed systems. In *SOSP*, 2015.

[122] Ménétrey, Jämes and Göttel, Christian and Khurshid, Anum and Pasin, Marcelo and Felber, Pascal and Schiavoni, Valerio and Raza, Shahid. Attestation Mechanisms for Trusted Execution Environments Demystified. In *DAIS*, 2022.

[123] Microsoft. Microsoft Power Automate. `https://flow.microsoft.com/`, 2020.

[124] Microsoft. Azure Confidential Computing. `https://azure.microsoft.com/en-us/solutions/confidential-compute`, 2020.

[125] Microsoft. Announcing: Microsoft moves $25 Billion in credit card transactions to Azure confidential computing. `https://techcommunity.microsoft.com/blog/azureconfidentialcomputingblog/announcing-microsoft-moves-25-billion-in-credit-card-transactions-to-azure-confi/3981180`, 2023.

[126] Microsoft. TPM Key Attestation - Microsoft. `https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/component-updates/tpm-key-attestation`, 2023. Accessed April 2025.

[127] Microsoft. Microsoft Azure Attestation. `https://learn.microsoft.com/en-us/azure/attestation/overview`, 2024.

[128] Microsoft. TPM Attestation in Azure. `https://learn.microsoft.com/en-us/azure/attestation/tpm-attestation-concepts`, 2024.

[129] Mohammad M. Ahmadpanah and Daniel Hedin and Musard Balliu and Lars Eric Olsson and Andrei Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security*, 2021.

[130] Mohammad M. Ahmadpanah and Daniel Hedin and Musard Balliu and Lars Eric Olsson and Andrei Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security*, 2021. URL `https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadpanah`.

[131] Mullender, Sape J. and van Rossum, Guido and Tanenbaum, Andrew S. and van Renesse, Robbert and van Staveren, Hans. Amoeba: A Distributed Operating System for the 1990s. *Computer*, 1990. URL `https://doi.org/10.1109/2.53354`.

[132] Musard Balliu and Iulia Bastys and Andrei Sabelfeld. Securing IoT Apps. *IEEE Security & Privacy Magazine*, 2019.

[133] Narayanan, Vikram and Carvalho, Claudio and Ruocco, Angelo and Almasi, Gheorghe and Bottomley, James and Ye, Mengmei and Feldman-Fitzthum, Tobin and Buono, Daniele and Franke, Hubertus and Burtsev, Anton. Remote attestation of confidential VMs using ephemeral vTPMs. In *ACSAC*, 2023.

[134] Needham, R M and Herbert, A J. *The Cambridge distributed computing system*. Addison Wesley Publishing Co., 1983. URL `https://www.osti.gov/biblio/5917199`.

[135] Phala Network. Phala Network Overview. `https://docs.phala.network/overview/phala-network`, 2022.

[136] Secret Network. Secret network overview - private smart con- tracts on the blockchain. https://scrt.network/about/about-secret-network/, 2022.

[137] Ousterhout, J.K. and Cherenson, A.R. and Douglis, F. and Nelson, M.N. and Welch, B.B. The Sprite network operating system . *Computer* , 1988. URL https://doi.ieeecomputersociety.org/10.1109/2.16.

[138] Parno, Bryan and Howell, Jon and Gentry, Craig and Raykova, Mariana. Pinocchio: nearly practical verifiable computation. In *IEEE S&P*, 2016. URL https://doi.org/10.1145/2856449.

[139] Pasquier, Thomas FJ-M and Singh, Jatinder and Bacon, Jean and Eyers, David. Information Flow Audit for PaaS Clouds. In *IC2E*, 2016.

[140] Pasquier, Thomas FJ-M and Singh, Jatinder and Eyers, David and Bacon, Jean. CamFlow: Managed data-sharing for cloud services. *IEEE Transactions on Cloud Computing*, 2017.

[141] Patrignani, Marco and Ahmed, Amal and Clarke, Dave. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys (CSUR)*, 51 (6), 2019.

[142] Pike, Rob and Presotto, Dave and Dorward, Sean and Flandrena, Bob and Thompson, Ken and Trickey, Howard and Winterbottom, Phil. Plan 9 from Bell Labs. *Comput. Syst.*, 8, 2000.

[143] Polinsky, Isaac and Datta, Pubali and Bates, Adam and Enck, William. SCIFFS : Enabling Secure Third-Party Security Analytics Using Serverless Computing. In *SACMAT*, 2021. URL https://doi.org/10.1145/3450569.3463567.

[144] Pubali Datta and Isaac Polinsky and Muhammad Adil Inam and Adam Bates and William Enck. ALASTOR: Reconstructing the Provenance of Serverless Intrusions. In *USENIX Security*, 2022.

[145] Puresec. FunctionShield. https://www.puresec.io/function-shield, 2018.

[146] Qiang, Weizhong and Dong, Zezhao and Jin, Hai. Se-Lambda: Securing Privacy-Sensitive Serverless Applications Using SGX Enclave. In *SecureComm*, 2018.

[147] Rabbani, Md Masoom and Vliegen, Jo and Winderickx, Jori and Conti, Mauro and Mentens, Nele. SHeLA: Scalable Heterogeneous Layered Attestation. *IEEE Internet of Things Journal*, 2019.

[148] AWS. AWS-codepipeline-stepfunctions. https://github.com/aws-samples/aws-codepipeline-stepfunctions, 2018.

[149] AWS Lab. Lambda Reference Architecture for Mapreduce. https://github.com/awslabs/lambda-refarch-mapreduce, 2018.

[150] Epsagon. Epsagon. https://epsagon.com/, 2018.

[151] Eric Brewer. gVisor: Protecting GKE and serverless users in the real world. https://tinyurl.com/gvisorsec, 2021.

[152] Google. gVisor. https://gvisor.dev/, 2021.

[153] Intrinsic. Intrinsic. https://intrinsic.com/, 2018.

[154] Jeremy Daly. Event Injection: Protecting your Serverless Applications. https://tinyurl.com/4udrdhmu, 2019.

[155] Nordstrom. Hello, Retail! https://github.com/Nordstrom/hello-retail, 2018.

[156] OpenFaas. OpenFaas. https://www.openfaas.com/, 2019.

[157] Ory Segal. Securing Serverless: Attacking an AWS Account via a Lambda Function. https://ubm.io/2FIrKq2, 2018.

[158] Patrick Wendell. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2019.

[159] Puresec. New Attack Vector: Serverless Crypto Mining. https://www.puresec.io/serverless-crypto-mining-resource-download, 2018.

[160] Vandium Software. Vandium. https://github.com/vandium-io, 2018.

[161] xmendez. Wfuzz. https://wfuzz.io/, 2021.

[162] zmq. Distributed Messaging - zeromq. http://zeromq.org/, 2019.

[163] Reynolds, Patrick and Killian, Charles Edwin and Wiener, Janet L and Mogul, Jeffrey C and Shah, Mehul A and Vahdat, Amin. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[164] Roy, Indrajit and Setty, Srinath TV and Kilzer, Ann and Shmatikov, Vitaly and Witchel, Emmett. Airavat: Security and Privacy for MapReduce. In *NSDI*, 2010.

[165] Sabelfeld, A. and Myers, A.C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003. doi: {10.1109/JSAC.2002.806121}.

[166] Sandy Schoettler and Andrew Thompson and Rakshith Gopalakrishna and Trinabh Gupta. Walnut: A low-trust trigger-action platform, 2020. https://arxiv.org/pdf/2009.12447.pdf.

[167] Sanjam Garg and Aarushi Goel and Somesh Jha and Saeed Mahloujifar and Mohammad Mahmoody and Guru-Vamsi Policharla and Mingyuan Wang. Experimenting with Zero-Knowledge Proofs of Training. Cryptology ePrint Archive, Paper 2023/1345, 2023. URL `https://eprint.iacr.org/2023/1345`.

[168] Sankaran, Arnav and Datta, Pubali and Bates, Adam. Workflow Integration Alleviates Identity and Access Management in Serverless Computing. In *ACSAC*, 2020.

[169] Daniel Schadt. hohibe.rs - Hierarchical Identity Based Encryption. `https://crates.io/crates/hohibe`, 2021. Accessed April 2025.

[170] Schuster, Felix and Costa, Manuel and Fournet, Cédric and Gkantsidis, Christos and Peinado, Marcus and Mainar-Ruiz, Gloria and Russinovich, Mark. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE S&P*, 2015. doi: {10.1109/SP.2015.10}.

[171] Sergei Arnautov and Bohdan Trach and Franz Gregor and Thomas Knauth and Andre Martin and Christian Priebe and Joshua Lind and Divya Muthukumaran and Dan O'Keeffe and Mark L. Stillwell and David Goltzsche and Dave Eyers and Rüdiger Kapitza and Peter Pietzuch and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 2016. URL `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov`.

[172] Serverless Computing. Serverless Computing. `https://en.wikipedia.org/wiki/Serverless_computing`, 2025. Accessed on June 23rd, 2025.

[173] Shahriar Ebrahimi and Parisa Hassanizadeh. From Interaction to Independence: zkSNARKs for Transparent and Non-Interactive Remote Attestation. In *NDSS*, 2024.

[174] Shepherd, Carlton and Markantonakis, Konstantinos and Jaloyan, Georges-Axel. LIRA-V: Lightweight Remote Attestation for Constrained RISC-V Devices. In *SPW*, 2021.

[175] Shixuan Zhao and Pinshen Xu and Guoxing Chen and Mengya Zhang and Yinqian Zhang and Zhiqiang Lin. Reusable Enclaves for Confidential Serverless Computing. In *USENIX Security*, 2023.

[176] Shweta Shinde and Dat Le Tien and Shruti Tople and Prateek Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS*, mar 2017. URL `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/panoply-low-tcb-linux-applications-sgx-enclaves/`.

[177] Shweta Shinde and Shruti Tople and Deepak Kathayat and Prateek Saxena. PodArch: Protecting Legacy Applications with a Purely Hardware TCB. Technical Report NUS-SL-TR-15-01, School of Computing, National University of Singapore, 2015.

[178] SiFive. SiFive U74 Core. `https://www.starfivetech.com/uploads/u74_core_complex_manual_21G1.pdf`, 2020.

[179] Sigelman, Benjamin H and Barroso, Luiz Andre and Burrows, Mike and Stephenson, Pat and Plakal, Manoj and Beaver, Donald and Jaspan, Saul and Shanbhag, Chandan. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google Inc., 2010.

[180] Sisi Duan and Hein Meling and Sean Peisert and Haibin Zhang. BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration. In *OPODIS*, 2014.

[181] Snyk. Snyk for Serverless. https://snyk.io/blog/launching-snyk-for-serverless/, 2017.

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

[183] Stumpf, Frederic and Fuchs, Andreas and Katzenbeisser, Stefan and Eckert, Claudia. Improving the Scalability of Platform Attestation. In *STC*, 2008.

[184] Surbatovich, Milijana and Aljuraidan, Jassim and Bauer, Lujo and Das, Anupam and Jia, Limin. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *WWW*, 2017.

[185] Technical City. Pentium Silver J5040 vs i7-10710U - Technical City. https://technical.city/en/cpu/Core-i7-10710U-vs-Pentium-Silver-J5040, 2019. Accessed April 2025.

[186] THL A29 Limited and Milo Yip. RapidJSON: A fast JSON parser/generator for C++ with both SAX/DOM style API. https://rapidjson.org/, 2016.

[187] Trach, Bohdan and Oleksenko, Oleksii and Gregor, Franz and Bhatotia, Pramod and Fetzer, Christof. Clemmys: Towards Secure Remote Execution in FaaS. In *SYSTOR*, 2019. URL https://doi.org/10.1145/3319647.3325835.

[188] Trusted Computing Group. TPM as an API for attestation in big, distributed environments. https://trustedcomputinggroup.org/tpm-as-an-api-for-attestation-in-big-distributed-environments/, 2022.

[189] Trusted Computing Group. CPU to TPM Bus Protection Guidance – Active Attack Mitigations. https://trustedcomputinggroup.org/wp-content/uploads/TCG_-CPU_-TPM_Bus_Protection_Guidance_Active_Attack_Mitigations-V1-R30_PUB-1.pdf, 2023.

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

[191] US Voter Breach. "database of 191 million u.s. voters exposed on internet.". https://www.reuters.com/article/us-usa-voters-breach-idUSKBN0UB1E020151229, oct "2015".

[192] Van Renesse, Robbert and Schneider, Fred. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, 2004.

[193] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016.

[194] Wagner, David A. Janus: an approach for confinement of untrusted applications. Master's thesis, University of California, Berkeley, 1999.

[195] Wagner, David and Dean, R. Intrusion Detection via Static Analysis. In *IEEE S&P*, 2001.

[196] Wang, Huibo and Wang, Pei and Ding, Yu and Sun, Mingshen and Jing, Yiming and Duan, Ran and Li, Long and Zhang, Yulong and Wei, Tao and Lin, Zhiqiang. Towards Memory Safe Enclave Programming with Rust-SGX. In *CCS*, 2019.

[197] Wang, Qi and Datta, Pubali and Yang, Wei and Liu, Si and Bates, Adam and Gunter, Carl A. Charting the Attack Surface of Trigger-Action IoT Platforms. In *CCS*, 2019. URL https://doi.org/10.1145/3319535.3345662.

[198] Will Glozer. wrk HTTP Benchmarking Tool. https://github.com/wg/wrk, 2012.

[199] WolfSSL. WolfSSL Embedded TLS Library. https://www.wolfssl.com/, 2012.

[200] WolfTPM. wolfTPM: Portable TPM 2.0 project designed for embedded use. https://github.com/wolfSSL/wolfTPM, 2023.

[201] xenbug. XSA-108: Improper MSR range used for x2APIC emulation. http://xenbits.xen.org/xsa/advisory-108.html, 2014.

[202] XMRig. XMRig Crypto Mining Software. https://xmrig.com/, 2017.

[203] Xu, Yuanzhong and Cui, Weidong and Peinado, Marcus. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE S&P*, 2015. doi: {10.1109/SP.2015.45}.

[204] Yan, Mengting and Castro, Paul and Cheng, Perry and Ishakian, Vatche. Building a Chatbot with Serverless Computing. In *MOTA*, 2016.

[205] Yehuda Lindell. Secure Multipaty Computation. https://cacm.acm.org/research/secure-multiparty-computation/, 2021.

[206] Youren Shen and Hongliang Tian and Yu Chen and Kang Chen and Runji Wang and Yi Xu and Yubin Xia and Shoumeng Yan. Occlum. In *ASPLOS*, mar 2020. URL https://doi.org/10.1145%2F3373376.3378469.

[207] Yunang Chen and Mohannad Alhanahnah and Andrei Sabelfeld and Rahul Chatterjee and Earlence Fernandes. Practical Data Access Minimization in Trigger-Action Platforms. In *USENIX Security*, aug

2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/chen-yunang-practical.

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

[209] Zapier. Zapier. https://zapier.com, 2020.

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

[211] Zeldovich, Nickolai and Boyd-Wickizer, Silas and Mazieres, David. Securing Distributed Systems with Information Flow Control. In *NSDI*, 2008.

[212] Zhao, Shijun and Zhang, Qianying and Qin, Yu and Feng, Wei and Feng, Dengguo. SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE. In *CCS*, 2019.

[213] Zhen Ling and Huaiyu Yan and Xinhui Shao and Junzhou Luo and Yiling Xu and Bryan Pearson and Xinwen Fu. Secure boot, Trusted boot and Remote attestation for ARM TrustZone-based IoT Nodes. *Journal of Systems Architecture*, 2021.