Guarding Serverless Applications with Kalium

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

Liang Wang
Princeton University
lw19@princeton.edu

Siddhant Bhagat
Microsoft
sbhagat3@wisc.edu

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

Abstract

As an emerging application paradigm, serverless computing attracts attention from more and more adversaries. Unfortunately, security tools for conventional web applications cannot be easily ported to serverless computing due to its distributed nature, and existing serverless security solutions focus on enforcing user specified information flow policies which are unable to detect the manipulation of the order of functions in application control flow paths. In this paper, we present *Kalium*, 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.

1 Introduction

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 [27, 38, 63].

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 [37, 44, 50]. 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 [5, 49, 56].

Previous work leverages information flow control (IFC) [3, 16] to solve the problem of **serverless data confidentiality** (See §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** (§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 §3.1).

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 [30]. 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.

The paper 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 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 used later in §3 to show the strengths of Kalium in protecting serverless data integrity.

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 §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 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.

2.3 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 [53] for regular programs. However, using IFC for serverless data integrity has not been explored in previous work. Trapeze [3] 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 [16] 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.

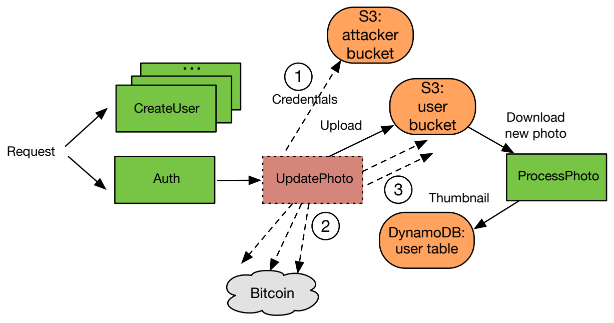


Figure 1: An example of serverless application. In the application, the `UpdatesPhoto` function uploads the photo sent by an authenticated user to AWS S3, which triggers the `ProcessPhoto` function to generate a thumbnail and store the thumbnail into AWS DynamoDB. The rectangle with dashed lines represents compromised functions, and dashed lines are flow injected by the adversary.

We show concrete attacks in §3.4.1 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 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.

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.

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 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 §3.4.1).

(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 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** [11, 44, 50]. 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 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 1 (2), the adversary performs her activity stealthily at the cost of the tenant. A concrete attack proposed by PureSec [50] 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 [11] that targets AWS Lambda to launch a variant of the XMRig [62] crypto-mining software has been discovered recently. SandTrap [2] 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 paper, we demonstrate that Kalium can defeat all three of the above-mentioned attacks.

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 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.

3.3 Challenges

To prevent the attacks discussed in §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 [46, 64]. 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

[1]We conducted this survey before the release of the Wonderless serverless application dataset [23].

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.

3.4 Limitations of existing approaches

3.4.1 Information flow control

IFC-based techniques for serverless data integrity have not yet been explored in previous work. Currently, IFC-based techniques [3, 16] 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.

3.4.2 Allow list based policies

Allow list based tools [15,20,34,35,49,58] 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 §3.4.1.

3.4.3 Log analysis techniques

Host-based intrusion detection systems (HIDS) detect potential attacks by monitoring an application's execution [18]. 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 [26]. The model is usually represented by an automaton. There is model-based HIDS research, focusing on model construction (dynamic analysis [24], static code analysis [59], static binary analysis [29], etc.) and model design (abstract stack model [59], Dyck model [36], inlining model [31], etc.). ALASTOR [17] 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.

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 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., [39]) 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 §5.1.

Adversary goals The adversary seeks to perform any type of control flow related attacks discussed in §3. We further assume that all the operations the adversary can perform must be done via the functions.

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, 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

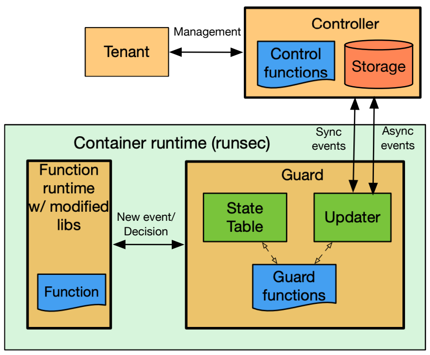


Figure 2: An overview of Kalium architecture.

network requests. We discuss flow graph generation and the guard and controller in this section.

5.1 Control flow tracking

All of the previously discussed attacks in §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 [1,29,36,59]. 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 $A \rightarrow B$) 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 [31]. Next, we discuss how to generate these graphs.

5.1.1 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 [16], 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 [10], 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 $\langle$function name, guard ID, request ID$\rangle$, 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 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 §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.

5.1.2 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.

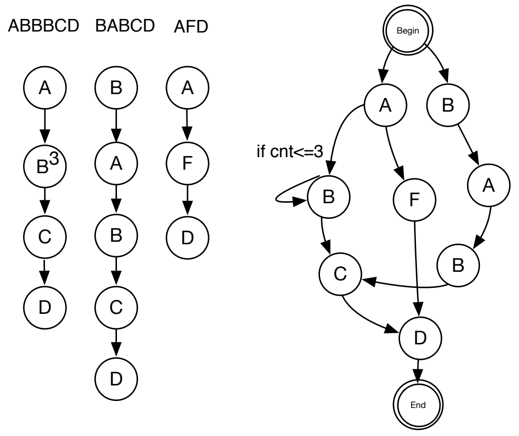


Figure 3: An example of the generated local flow graph for three traces ABBBCD, BABCD, and AFD.

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 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 ($f' \to s \to f$) 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 §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.

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 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 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 ACM-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

`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 4: runsec API and Userspace Hypercall API

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 §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.

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 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.

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. We demonstrate the use of Kalium for rate-limiting in Appendix §A. However, in this paper we focus on the enforcement of control-flow integrity over HTTP and TLS requests.

6 Kalium Implementation

Secure runtime for serverless functions. The Kalium container runtime runsec is built atop gVisor [30], 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 [21].

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* [65]. 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 (§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.

7 Evaluation

We evaluate the security and performance overhead of Kalium. We run our implementation of Kalium in a local testbed running OpenFaas [43] 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.

7.1 Workloads

We evaluate Kalium on three sets of workloads:

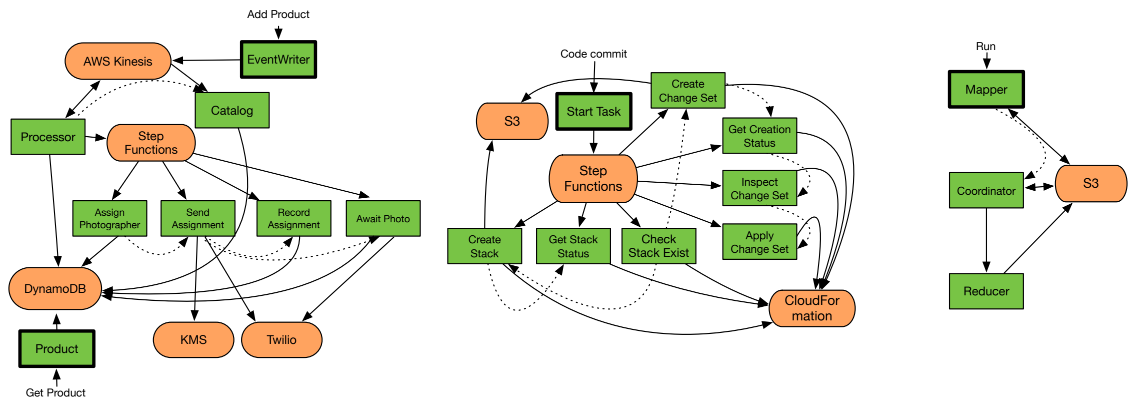
Wonderless serverless application dataset. We started with the Wonderless Dataset for Serverless Computing [23], which comprises all open source applications scraped from all public GitHub repositories. As there were *no* OpenFaaS-based applications in the dataset, we port the OpenWhisk-based applications 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 [42], CodePipeline [6], and MapReduce [7]. 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 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 6 and Figure 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 [16]. 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.

7.2 Efficiency of flow tracking

Local testbed setup. We set up a single control plane Kubernetes cluster of five nodes in CloudLab [19], 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



(a) HelloRetail (partial) (b) CodePipeline (c) MapReduce

Figure 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.

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

Table 6: An overview of the lines of code, number of functions and third-party modules, and the languages of the target applications.

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 [61] 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 [47], 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.

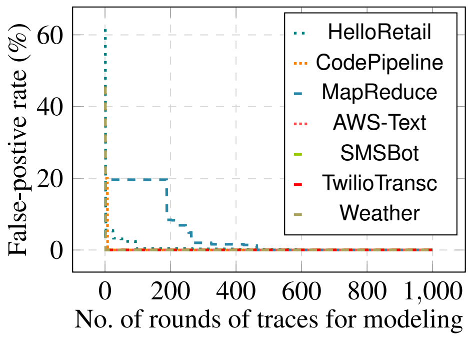


Figure 7: False-positive rates when using the traces collected from different numbers of rounds for building flow graphs.

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

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 [28, 60] 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 §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.

7.4 Performance overhead of Kalium

We evaluate performance of Kalium on the local testbed using the benchmark from Valve [16].

Performance overhead comparison of Kalium, Valve and Trapeze. We first run the benchmark suite used by Valve [16] 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 ($\leq$ 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 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 (§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 (§5.2), (c) event

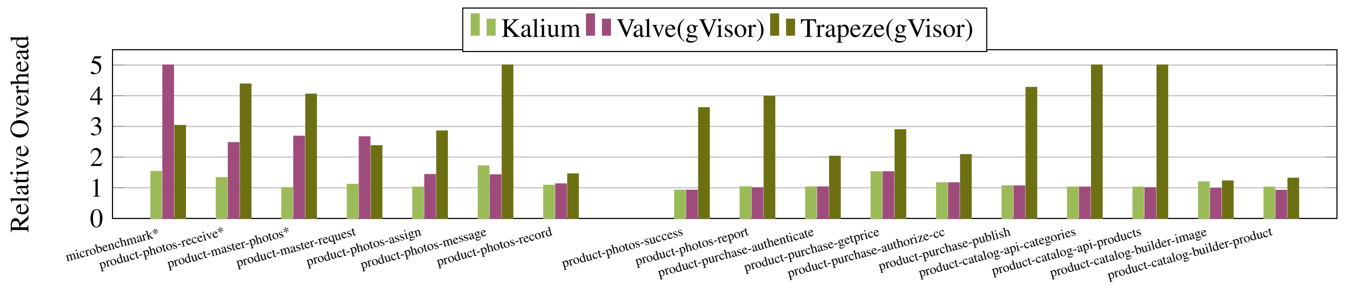


Figure 8: The relative latency overhead of the functions in HelloRetail and the microbenchmark function. Valve proxies network requests only in functions marked with an asterisk. Y-axis is truncated at 5.

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 (§8).

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 (§7.2, §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 [4] to yield more accurate flow graphs.

Concurrent requests. As mentioned earlier (§5.1.2), 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 [12] 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.

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 [8, 45, 46, 52, 64], so, naturally, it can be used for improving serverless security. Trapeze [3] 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 [9, 13,14,22,25,32,33,40,41,51,55]. 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 [54] 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 [48] 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 [57] 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.

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.

Acknowledgement

We would like to thank Tom Ristenpart, Somesh Jha, Earlence Fernandes, and the anonymous shepherd and reviewers for very useful feedback and comments. We would also like to thank all the members of the SCAIL group at the University of Wisconsin Madison and Sambhav Satija for feedback on earlier drafts of the paper. This work was supported by NSF grant CNS 1763810.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS '05*, pages 340–353. ACM, 2005.

[2] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. Sandtrap: Securing javascript-driven trigger-action platforms. In *USENIX Security '21*, pages 2899–2916, 2021.

[3] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. *arXiv preprint arXiv:1802.08984*, 2018.

[4] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, nov 1987.

[5] AquaSec. Security for serverless functions. `https://snyk.io/blog/launching-snyk-for-serverless/`, 2017.

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

[7] AWS Lab. Lambda reference architecture for mapreduce. `https://github.com/awslabs/lambda-refarch-mapreduce`, 2018.

[8] Jean Bacon, David Eyers, Thomas FJ-M Pasquier, Jatinder Singh, Ioannis Papagiannis, and Peter Pietzuch. Information flow control for secure cloud computing. *IEEE Transactions on Network and Service Management*, 11(1):76–89, 2014.

[9] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI '04*, volume 4, pages 18–18, 2004.

[10] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE '11*, pages 267–277, 2011.

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

[12] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security '15*, page 161–176, 2015.

[13] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, page 595. IEEE, 2002.

[14] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI '08*, volume 8, pages 117–130, 2008.

[15] Cilium. Cilium: Container observability using ebpf. https://cilium.io/, 2019.

[16] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing Function Workflows on Serverless Computing Platforms. In *WWW '20*, pages 939–950, 2020.

[17] Pubali Datta, Isaac Polinsky, Muhammad Adil Inam, Adam Bates, and William Enck. ALASTOR: Reconstructing the provenance of serverless intrusions. In *USENIX Security '22*, pages 2443–2460, 2022.

[18] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, 1987.

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *USENIX ATC '19*, pages 1–14, 2019.

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

[21] Eric Brewer. gvisor: Protecting gke and serverless users in the real world. https://tinyurl.com/gvisorsec, 2021.

[22] Robert Escriva, Ayush Dubey, Bernard Wong, and Emin Gün Sirer. Kronos: The design and implementation of an event ordering service. In *EuroSys '14*, page 3. ACM, 2014.

[23] Nafise Eskandani and Guido Salvaneschi. The wonderless dataset for serverless computing. In *MSR '21*, pages 565–569, 2021.

[24] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *IEEE S&P '03*, pages 62–75. IEEE, 2003.

[25] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *NSDI '07*, pages 20–20. USENIX Association, 2007.

[26] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for unix processes. In *IEEE S&P '96*, pages 120–128. IEEE, 1996.

[27] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI '17*, pages 363–376, 2017.

[28] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS '03*, 2003.

[29] Jonathon T Giffin, Somesh Jha, and Barton P Miller. Detecting manipulated remote call streams. In *USENIX Security '02*, pages 61–79, 2002.

[30] Google. gvisor. https://gvisor.dev/, 2021.

[31] Rajeev Gopalakrishna, Eugene H Spafford, and Jan Vitek. Efficient intrusion detection using automaton inlining. In *IEEE S&P '05*, pages 18–31. IEEE, 2005.

[32] Xueyuan Han, Thomas Pasquier, and Margo Seltzer. Provenance-based intrusion detection: Opportunities and challenges. *arXiv preprint arXiv:1806.00934*, 2018.

[33] Md Nahid Hossain, Junao Wang, Ofir Weisse, R Sekar, Daniel Genkin, Boyuan He, Scott D Stoller, Gan Fang, Frank Piessens, Evan Downing, et al. Dependence-preserving data compaction for scalable forensic analysis. In *USENIX Security 2018*, pages 1723–1740, 2018.

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

[35] Istio. Istio service mesh for kubernetes. https://istio.io/, 2019.

[36] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *NDSS '04*, 2004.

[37] Jeremy Daly. Event injection: Protecting your serverless applications. https://tinyurl.com/4udrdhmu, 2019.

[38] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *SoCC '17*, pages 445–451. ACM, 2017.

[39] Zhen Ling, Junzhou Luo, Yang Zhang, Ming Yang, Xinwen Fu, and Wei Yu. A novel network delay based side-channel attack: Modeling and defense. In *IEEE INFOCOM 2012*, pages 2390–2398, 2012.

[40] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS '18*, 2018.

[41] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *SOSP '15*, pages 378–393. ACM, 2015.

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

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

[44] Ory Segal. Securing serverless: Attacking an aws account via a lambda function. https://ubm.io/2FIrKq2, 2018.

[45] Thomas FJ-M Pasquier, Jatinder Singh, Jean Bacon, and David Eyers. Information flow audit for paas clouds. In *IC2E '16*, pages 42–51. IEEE, 2016.

[46] Thomas FJ-M Pasquier, Jatinder Singh, David Eyers, and Jean Bacon. Camflow: Managed data-sharing for cloud services. *IEEE Transactions on Cloud Computing*, 5(3):472–484, 2017.

[47] Patrick Wendell. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2019.

[48] Isaac Polinsky, Pubali Datta, Adam Bates, and William Enck. SCIFFS : Enabling secure third-party security analytics using serverless computing. In *SACMAT '21*, page 175–186, 2021.

[49] Puresec. Functionshield. https://www.puresec.io/function-shield, 2018.

[50] Puresec. New attack vector: Serverless crypto mining. https://www.puresec.io/serverless-crypto-mining-resource-download, 2018.

[51] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI '16*, volume 6, pages 9–9, 2006.

[52] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and Privacy for MapReduce. In *NSDI '10*, volume 10, pages 297–312, 2010.

[53] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[54] Arnav Sankaran, Pubali Datta, and Adam Bates. Workflow integration alleviates identity and access management in serverless computing. In *ACSAC '20*, pages 496–509, 2020.

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

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

[57] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards Secure Remote Execution in FaaS. In *SYSTOR '19*, page 44–54, 2019.

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

[59] David Wagner and R Dean. Intrusion detection via static analysis. In *IEEE S&P '01*, pages 156–168. IEEE, 2001.

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

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

[62] XMRig. Xmrig crypto mining software. https://xmrig.com/, 2017.

[63] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *MOTA '16*, page 5. ACM, 2016.

[64] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing distributed systems with information flow control. In *NSDI '08*, page 293–308, 2008.

[65] zmq. Distributed messaging - zeromq. http://zeromq.org/, 2019.

A Rate-limiting functions

We use Kalium to develop another function for a conventional security task to demonstrate it's flexibility: The *rate limiting* function restricts the number of instances of a given function running in a certain time period and the rates of given API requests generated from a function to mitigate DoS attacks against the application and prevent the application from being used for DoS.

We consider two types of rate limiting tasks: (1) limiting the fan-out (i.e., the number of running instances) of a given function, and (2) limiting requests to a given API. For (1), since every guard will register with the controller during startup, it is easy for the controller to maintain a per-function counter to track the number of launched instances for a given function. However, the provider might launch new instances to execute functions, and discard the existing instances. The controller may see a large number of new instances over time but only a portion of them are active. To handle this case, the controller will check if the instances are active by periodically (i.e., every second) sending heartbeat messages, and decrease the corresponding counters for those instances that are no longer active.

Using the controller as a centralized state store, we can restrict the frequency of requests sent to a given URL for a given operation from a function of interest. A rate limiting policy is a list of entries <function name, target URL, HTTP

operation, rate$>$. Once a tenant deploys the rate limiting functions and policies on the guard and the controller, the controller will keep track of the events from functions, and calculate the rate of a given type of flow. If the rate of the flow exceeds the allowed rate, the controller will broadcast a STOP event to all the guards that are responsible for the functions generating the flow. Then, the guards will simply drop the flow, whose type is specified in the STOP event, until receiving a RESUME event from the controller.