Growlithe: A Developer-Centric Compliance Tool for Serverless Applications

Praveen Gupta, Arshia Moghimi, Devam Sisodraker, Mohammad Shahrad, Aastha Mehta
The University of British Columbia
pvgupta@cs.ubc.ca, amoghimi@student.ubc.ca, devam@student.ubc.ca, mshahrad@ece.ubc.ca, aasthakm@cs.ubc.ca

***Abstract*—Serverless applications consist of functions written in heterogeneous programming languages, use diverse data stores and communication services, and evolve rapidly. Consequently, it is challenging for serverless tenants to protect their application data from inadvertent leaks due to bugs, misconfigurations, and human errors. Cloud security tools, such as Identity and Access Management (IAM), lack observability into a tenant's application, whereas the state-of-the-art dataflow tracking tools require support from the cloud platform and incur significant runtime overheads. We present Growlithe, a tool that integrates with the serverless application development toolchain and enables continuous compliance with data policies by design. Growlithe allows declarative specification of access and data flow control policies over a language- and platform-independent dataflow graph abstraction of a serverless application, and enforces these policies through a combination of static analysis and runtime enforcement. We used Growlithe with applications using Python and JavaScript functions that can be hosted on AWS Lambda and Google Cloud Functions platforms. We empirically demonstrate that Growlithe is cross-cutting, portable and efficient, and enables developers to easily adapt their application and policies to evolving requirements.**

1. Introduction

Security and compliance is a shared responsibility between cloud providers (e.g., AWS, Azure, Google Cloud) and their tenants [1–4]. In light of the recent data privacy regulations, such as GDPR [5], CCPA [6], CPPA [7], the responsibility of ensuring compliance with the regulations remains with the cloud tenant (the data controller in GDPR parlance[1]), whereas the cloud platform provider (the data processor) is expected to only process the data according to the tenant's instructions. Cloud providers offer mutual isolation for tenants' code, e.g., through virtual machines, containers, and trusted execution environments (e.g., SGX enclaves), and data, e.g., through identity and access management tools (IAM) [8–10], network access control [11]), and encryption for data in transit. However, a remaining challenge is protecting a tenant's data against its own bugs, vulnerabilities, and misconfigurations.

Several tools were designed in the last few decades for securing data in monolithic applications [12–19]. These tools explored language designs for specifying the desired data policies and system implementations for enforcing the policies through access control, and static and dynamic information flow control mechanisms. However, organizations are increasingly migrating their applications to serverless platforms (e.g., AWS Lambda, Microsoft Azure Functions, Google Cloud Functions) [20], which offer auto-scaling, high availability, and a pay-per-use model, enabling developers to focus on their business logic. Unfortunately, the pre-serverless era tools cannot address the challenges in ensuring compliance that are pertinent to the serverless computing paradigm and platforms.

First, unlike prior work, a tool for serverless applications needs to handle heterogeneity within the application, which arises from functions written in diverse programming languages (e.g., JS, Python, C, Go) and relying on diverse services (e.g., queues, key-value stores, and databases) [20]. Second, unlike conventional long-running applications, serverless functions typically have very short startup and execution times (in the order of milliseconds to a couple of seconds) [21], and even the inter-function communication over the network adds to the application overheads. Thus, a compliance tool for serverless must minimize runtime overhead to avoid impacting the overall application performance and ultimately its billing costs. Third, a tool needs to address challenges due to the serverless *platform*, such as the possibility of leakage of data across client requests due to the container reuse optimization designed for reducing performance overheads. Finally, while most prior work focuses on evaluating their tool on a snapshot of an application and its data, an effective tool must be able to keep up with frequent changes in a serverless application. In particular, different parts of the application, which may be maintained by different development teams, may evolve independently of each other. Ensuring continued end-to-end compliance in such an evolving application is non-trivial.

Cloud IAM tools [8–10] allow cloud tenants to restrict access to their applications and resources at a coarse granularity (e.g., allowing read/write access on a database to a specific function or a tenant). However, IAM tools are not sufficient for tenants to ensure compliance for their users' data records, which would require information flow control. Recent work has proposed new tools for cloud tenants to specify and enforce data flow restrictions in serverless applications both at a coarse granularity (similar to IAM) [22, 23] and at a fine granularity (accounting for user-specific policies) [24]. However, these tools either support applications written in a single programming language, incur significant

1. Other regulations [6] also define equivalent roles.

runtime overhead, or depend on custom containers, thus requiring support from cloud providers for adoption.

We present Growlithe, a tool that serverless application developers can integrate in their software development toolchain and achieve compliance without requiring any modifications or support from cloud providers. Growlithe requires modest developer inputs for specifying the policies required by the users, the organization, or the regulations, and enables efficient policy enforcement at the application source code layer.

Growlithe builds a language- and platform-independent dataflow graph abstraction of a serverless application (§3.1), on which developers specify data policies, independent of the application implementation. Developers specify fine-grained access control and information flow control policies in a declarative specification language, which capture requirements for storing and processing sensitive data of the applications' end users (§3.2).

Growlithe enforces the policies through a combination of static and dynamic enforcement (§3.4). The static enforcement runs on the abstract application dataflow graph and, therefore, is language- and platform-independent. For runtime enforcement, Growlithe instruments the application source code with assertions that are checked during application execution. The runtime checks are also implemented largely in a platform-independent manner, though Growlithe may leverage platform-specific tools (e.g., logging services) for performance optimization. Static enforcement helps to reduce performance overheads of the runtime enforcement, while runtime enforcement helps to enforce the policies that could not be checked statically.

Growlithe logs policy check failures during offline and runtime enforcement with information about the inputs and the operations involved in the policy violations. Developers can use these logs to revise their application code, configuration, or policy specification, and re-run Growlithe before deploying the new version of the application.

Contributions. To the best of our knowledge, Growlithe is the first tool designed to empower developers in enabling *continuous policy compliance by design* within their serverless applications. Growlithe centralizes specification of data policies for serverless applications consisting of one or more functions, data stores, and communication services, and automates the enforcement of the policies. Together, these two features help to ease the developer's burden in ensuring policy compliance.

Concretely, we make the following contributions. (i) We present language- and platform-independent abstractions, a declarative policy specification language, and a hybrid policy enforcement mechanism with a narrow interface for integration with function implementations (§3). (ii) We provide a prototype of Growlithe, which supports Python and JavaScript functions (the two most popular languages for serverless [20]), and can be run on AWS Lambda and Google Cloud Functions (§4). (iii) We demonstrate Growlithe's generality and efficacy in enabling compliance by applying it to three applications, covering different characteristics of realistic serverless application architectures (§5.1). (iv) We quantify Growlithe's performance costs (§5.2, §5.3), analyze the costs involved in supporting new programming languages and services and in porting to a different cloud platform (§5.4), and discuss Growlithe's effectiveness in addressing policy violations due to bugs or evolving application requirements (§5.5).

2. Motivation and Goals

We first motivate the need for a new security tool for serverless applications, then we discuss the high-level goals of this tool and the threat model it targets.

2.1. A Primer on Serverless Paradigm

Serverless computing is a new paradigm, in which applications are architected as a collection of multiple stateless functions that are run as independent units of computation, rely on external data stores for persisting their data, and communicate with each other via network or external data stores. Cloud platforms run these functions in containers, manage the allocated resources, and automatically scale them to adapt to variable loads.

Protecting data in a serverless application requires understanding where the data is stored and accessed, and how the data is communicated between different components of the application. Communication to/from functions can occur in several ways. First, the functions may communicate with each other via data stores (e.g., databases, key-value stores, file systems), or communication services (e.g., message queues or broker services). Second, a function may directly invoke another function (e.g., using a remote procedure call, or REST APIs), where data may be passed through arguments to the invoked function and results returned from the called functions. Third, orchestration services [25] may mediate routing of the control and results from one function to another according to the control flow configuration of the application. Functions may also call other external services, e.g., data store services, by invoking the public APIs of the services. Similarly, functions may be invoked in response to specific "event triggers" which may originate from external services, client requests to the functions, or periodic events.

Additionally, data may flow between different invocations of a function that end up executing in the same "warm" containers, which are reused to reduce function startup times. Since container scheduling policies are opaque to serverless applications, it is challenging to predict which dataflows across invocations may be potentially sensitive.

In this paper, we propose Growlithe to enforce constraints on such data access and data flow patterns to achieve compliance with user-defined data policies.

2.2. Motivating Example

Consider the example of a fictitious insurance company *i*Claim, which runs a claim processing application called ClaimApp as a serverless application. Several companies

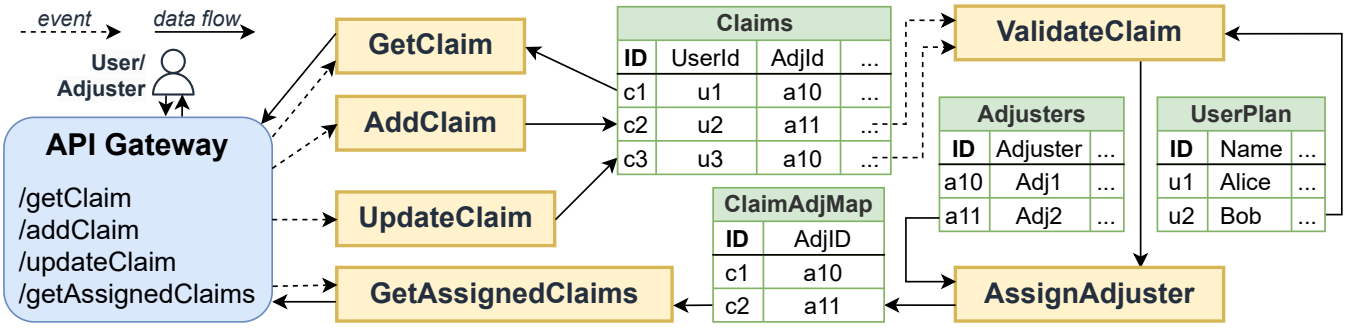

Figure 1: A sample insurance claim processing app. Functions are triggered by APIs, invocations, and DynamoDB events.

already use serverless workflows for insurance services [26–28]. Figure 1 shows the application architecture and workflows. ClaimApp serves two types of end users: the insurance customers, who submit claims and check their claims' status, and the company's employees or insurance adjusters who manually verify the submitted claims. The application maintains four tables: `UserPlan`, which contains customers' coverage plans, `Claims`, which contains the claims submitted by customers, `Adjusters`, which contains adjusters' details, and `ClaimAdjMap`, which maps claims to the adjuster assigned for verification. The application exposes four API endpoints. Three of them allow a user to add a new claim (`AddClaim`), update an existing claim (`UpdateClaim`) in `Claims`, and retrieve a claim (`GetClaim`) from `Claims`. The fourth endpoint allows an adjuster to retrieve the list of claims assigned to her from `ClaimAdjMap`.

The application backend contains two additional functions `ValidateClaim` and `AssignAdjuster`. The function `ValidateClaim`, which is triggered upon adding or updating a claim in `Claims`, performs preliminary checks against the claimant's coverage plan and, if the claim is valid, invokes `AssignAdjuster`. The function `AssignAdjuster` assigns an adjuster to an unassigned claim by adding a mapping in `ClaimAdjMap`.

We consider three policies that the insurance company may wish to enforce on their customers' data.

P1 Customers should only be able to access and update their own claims.

P2 Adjusters should only be able to access and process claims that are assigned to them.

P3 Customers' plan data should only be used for validating submitted claims and not be accessible to adjusters.

These policies ensure the confidentiality and integrity of users' claims and plan details. However, enforcing these policies can be challenging. For instance, for policy **P3**, developers should ensure that there is no data flow from `UserPlan` table read in `ValidateClaim` to the `ClaimsAdjMap` table written in the subsequent `AssignAdjuster` function. In general, complying with such policies requires enforcing per-user record access control and data flow control, which the state-of-the-art solutions [8–10, 22] do not support.

We use this application as a running example throughout this paper to explain different parts of Growlithe's design.

2.3. Design Goals

Growlithe is designed to address the following goals.

D1. It must be able to protect data in the presence of multiple complex workflows in an application, i.e., it must be able to enforce complex policies involving data accesses and data flows within and across workflows.

D2. Given the complexity of serverless applications and policies, the tool must ease the developers' burden of specifying and reasoning about the correctness of the policies.

D3. The tool should be easy to port across different cloud platforms to increase adoption for multiple serverless applications and avoid vendor lock-in.

D4. The tool must be able to handle application components written in different programming languages and relying on diverse data store and communication services.

D5. The tool must handle the cloud platform's scheduling and resource management semantics (e.g., container warm-start), which may impact the application's data flow across components.

2.4. Threat Model

Growlithe's goal is to enable serverless developers to protect their own data against inadvertent leaks and improper use. We assume that the developers are not malicious and wish to secure users' and applications' data. However, their application is susceptible to compliance violations due to bugs and misconfigurations, which may arise due to complexity of the application's workflows, as well as the platform's underlying scheduling and resource management semantics that are typically hidden from developers in serverless settings (e.g., reusing "warm" containers for future function invocations).

Growlithe's compliance jurisdiction is the application code and boundaries, and it does not inspect external services or functions. For example, if an application relies on a cloud database service, Growlithe only covers the APIs exposed to the application, but not the database internals. If the

Id name	Description
Data conduit identifiers	
Object	Reference to an event (e.g., trigger) or an object being accessed (e.g., an S3 file, or a DynamoDB record, function parameters).
ObjType	Type of the event or object being accessed.
ObjAttrs	Extensible list of attributes associated with a given object (e.g., ContentType, LastModified)
Resource	Reference to the base resource of an object (e.g., S3 bucket, DynamoDB table).
ResourceAttrs	Extensible list of attributes associated with the base resource (e.g., ResourceRegion, indicating the region in which the resource is deployed).
ObjFn	Name of the function where the object is accessed.
ObjCodeLoc	Function source file, line where the object is accessed.
Conduit API identifiers	
ObjReadAPIs	Set of APIs for reading the object (e.g., `GetItem`, `Query`, `Scan` for DynamoDB).
ObjWriteAPIs	Set of APIs for writing to the object (e.g., `PutItem`, `UpdateItem` for DynamoDB).
Function instance identifiers	
InstName	Name of the serverless function instance deployed.
InstRegion	Deployment region of the function instance.
InstTime	Time at which the function instance makes a request.
User session identifiers	
SessionAuth	Session authentication credentials of the end user making a request.
SessionRegion	Region from where an end user makes a request.
Dataflow graph identifiers	
CurrNode	Reference to the current node.
PredNode	Reference to the predecessor node.

TABLE 1: Growlithe's abstract identifiers.

service is considered un-trustworthy, the application could send data to the service in encrypted form and decrypt the data retrieved from the service only within the application.

Growlithe's guarantees depend on two things. First, the developer must specify correct policies and create the correct mapping of the information used in policy evaluation (§3). Second, the cloud provider must be trusted to fulfill its share of responsibilities towards its customers for ensuring security and compliance [2–4]. Thus, we assume that the cloud platform is correct (e.g., orchestrates functions correctly), secured (e.g., no exploits in the function containers), and does not attack or bypass Growlithe (e.g., deploys the correct functions containing Growlithe's instrumentation).

All active attacks against the tenants' applications, the cloud platform and Growlithe, and the mitigations for the attacks are orthogonal to this work.

3. Growlithe Design

Growlithe is a tool that integrates with a serverless application development lifecycle to enable *compliance by design*. In order to minimize cloud-platform and language dependencies, Growlithe first defines five types of identifiers (Table 1), which it uses in various abstractions, namely, a dataflow graph, the policy specification language, and the taint tracking infrastructure. While integrating Growlithe with an application deployed on a specific cloud platform, developers only need to specify the mappings of these identifiers to variables in application source code or values from the cloud platform environment, which enables Growlithe to perform compliance checks in the context of the specific instantiation of the application on the platform.

Once the mappings are defined, Growlithe consists of four operational stages, as shown in Figure 2: dataflow graph generation, policy specification, static enforcement, and runtime enforcement. The first three stages operate on an application prior to its deployment, while the last stage operates on the deployed application.

Dataflow Graph Generation. ① Growlithe analyzes the application source code and configurations and generates a dataflow graph. The analysis is done in a modular fashion. Growlithe performs a per-function dataflow analysis on each serverless function independently. Additionally, with the help of the configuration input, Growlithe identifies the application workflows and performs an inter-function dataflow analysis along those workflows. It combines the dataflow graphs of all the analyses to generate the complete application dataflow graph (ADG).

Policy Specification. ② Growlithe automatically generates policy templates based on the ADG. The templates show the data flows discovered in the application workflows and denies all the flows. If any policies were already defined for the application previously, Growlithe adds them to the newly generated templates (see §3.2 for policy semantics). ③ Optionally, a developer can further refine the generated policy templates manually based on specific compliance requirements. ④ Growlithe adds appropriate policies to the edges of the ADG to generate an annotated application dataflow graph ($\overline{ADG}$).

Static Enforcement. ⑤ Next, Growlithe statically checks the policies on the annotated dataflow graph. When Growlithe encounters a policy check failure on a dataflow path, it generates an error, and logs the dataflow path with the annotated policies and the inputs, which helps developers to debug the failure and revise their application or policy specification. ⑥ For policies that depend on dynamic inputs (e.g., time of day, dynamic reference to a resource), Growlithe defers the checks to runtime. For this, it adds the corresponding policies as assertions in the application source code at appropriate places identified by the edges of the dataflow graph. Growlithe also adds taint propagation logic in the functions and generates IAM policies that allow basic execution of the functions with all accesses to the application's resources, which is required to allow deployment of the application.

Runtime Enforcement. ⑦ The instrumented application code is deployed along with the configuration and IAM policies. ⑧ As the application executes different workflows, the taint propagation logic and assertions help to enforce the deferred policies. When an assertion fails, Growlithe stops

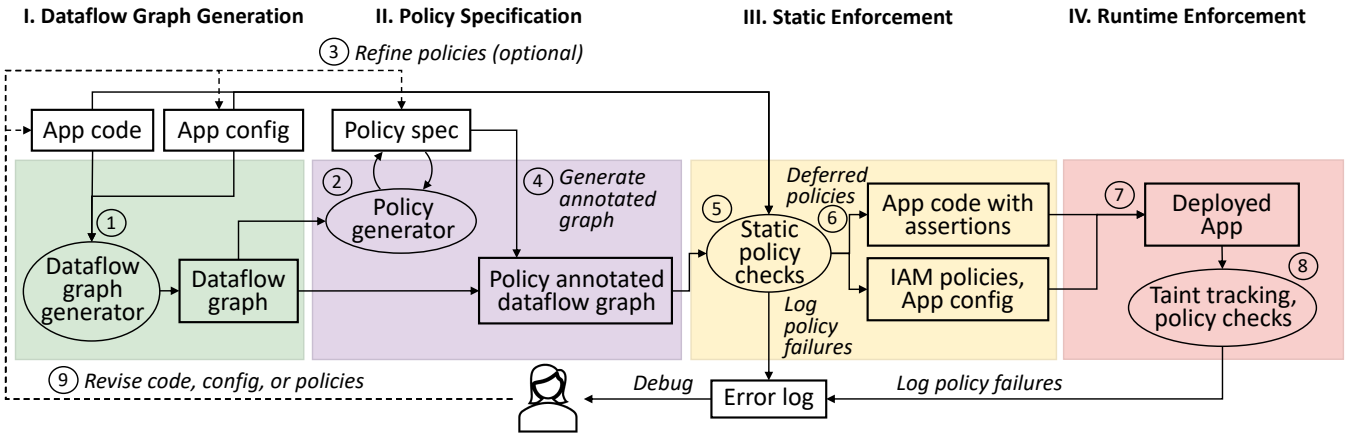

Figure 2: Growlithe's operational stages integrated with the serverless application development lifecycle.

the execution of the function and logs the event with the details of the offending dataflow path, assertion, and inputs.

⑨ Developers can inspect the logs for policy violations and revise the source code, configurations, or policy specifications for the next version of the application.

We now elaborate on these stages and explain Growlithe's design.

3.1. Dataflow Graph Generation

For the offline stages, Growlithe relies on a language-independent dataflow graph abstraction of the application, on which it specifies policies and enforces these policies statically. We define Growlithe's application dataflow graph as a directed acyclic graph, $ADG = (V, E)$, where V is the set of nodes representing *data conduits*, i.e., entities that contain or carry data to be subjected to compliance policies, and $E \subseteq V \times V$ is the set of directed edges representing the flow of data between the data conduits.

Data flows between serverless functions and to/from external services in several ways as indicated by the different communication modes described in §2.1. For each of these communication modes, Growlithe models the data conduits, such as function arguments, return objects, file systems, queues and other data stores.

An edge between two conduit nodes is represented as $e = (v_1, v_2, \pi) : v_1, v_2 \in V, v_1 \neq v_2$, which implies that data is *read* from the *source* conduit v_1 and *written* to the *sink* conduit v_2. For each conduit type, Growlithe models the associated APIs into read and write operations. The label π encodes edge properties, e.g., the type of the edge ($\pi.type$) and the policy on the edge ($\pi.policy$). The edge type indicates an intra-function or an inter-function dataflow. A policy consists of conditions under which data can be read from the source conduit and written to the sink conduit. We discuss the policy specification in detail in §3.2.

Per-function dataflow graph generation. Growlithe generates ADG in two steps. In the first step, it generates a dataflow graph for each function of a serverless application. For this, it uses CodeQL [29], which takes as input the source code of a function f, the set of source conduits S_f, and the set of sink or target conduits T_f, and generates an AST (abstract syntax tree), a CFG (control flow graph), and the data dependencies between the function elements based on define-use relationships [30]. (We discuss Growlithe's CodeQL module further in §4.)

Besides the standard define-use rules that capture read-after-write dependencies in conventional program syntax (e.g., assignment statements, conditional statements, etc.), Growlithe needs to capture read-after-write dependencies across data conduit APIs to track data flows. For example, a serverless function may write to a temporary file in its container's local file system (or a DB record) and subsequently read the file (or the record) in the same or a different function invocation. Developers model data flow rules between the read and write conduit API identifiers defined in Table 1.

While many conduits have distinct read and write APIs, some APIs in dynamic languages like JavaScript may have complex, unconventional semantics. For example, Amazon's S3 service provides an API on an S3 bucket with the following signature: `bucket.download_file('REMOTE_OBJECT', 'LOCAL_FILE')`, which allows a function to download `REMOTE_OBJECT` from its S3 bucket and store it in `LOCAL_FILE` in the temporary file system of the container hosting the function. In this case, the data flow between `bucket.REMOTE_OBJECT` and `LOCAL_FILE` is encoded within a single API's semantics. Developers may manually model the data flow rules for such APIs for each conduit in each language used in their application.

Growlithe combines the IR abstractions along with the sets S_f and T_f to generate a per-function dataflow graph. For each node in the dataflow graph, Growlithe records a uniquely generated node identifier, as well as one or more conduit properties parsed from the function source code. The data conduit identifiers in Table 1 are the properties recorded by Growlithe for the dataflow graph nodes.

Linking per-function dataflow graphs. In the second step, Growlithe parses the application's configuration files, which provide information on inter-function dependencies and the sequence of function invocations. Growlithe uses this information to link the per-function dataflow graphs to generate the end-to-end application dataflow graph ADG.

Cross-language ADG. Building the ADG across components written in multiple languages requires some language-specific modelling. Specifically, a developer needs to model data conduits and the read/write operations on the data conduits in each language. This is a one-time effort required from a developer for each conduit type in each language (see more discussion in §5.4).

Example. Figure 3 shows a subset of the final ADG generated by Growlithe for the claim processing application (cf. §2.2) written in Python for the AWS lambda platform. Growlithe first generates dataflow graphs for each lambda function in the application. $G_{vc} = (V_{vc}, E_{vc})$ and $G_{aa} = (V_{aa}, E_{aa})$ for the functions `ValidateClaim` and `AssignAdjuster`, respectively. The vertex and edge sets for the graphs are as follows: $V_{vc} = \{n_1, n_2, n_3\}$, $E_{vc} = \{e_1, e_2\}$, $V_{aa} = \{n_4, n_5, n_6\}$, and $E_{aa} = \{e_4, e_5\}$. The edge e_3 represents an inter-function data flow.

3.2. Policy Specification Framework

Once the ADG is generated, Growlithe associates policies with the graph edges to generate an annotated graph, $\overline{ADG} = (V, E')$, where $E' = \{(s_i, d_i, \pi_i') : (s_i, d_i) \in E, \pi_i' = (\pi_i.type,\ \pi_i.\text{policy} \neq \phi)\}$. We now describe Growlithe's policy specification language and the edge policy semantics.

Growlithe uses a declarative policy specification language, which is similar to Datalog [31] and inspired from the specification defined in Thoth [16]. An edge policy consists of two types of permissions: read and write, which specify conditions under which data can be read from the source conduit and written to the sink conduit of the edge, respectively. These conditions are expressed as one or more *clauses* connected by a disjunction (i.e., or, written "∨"), where each clause further consists of one or more *predicates* connected by conjunction (i.e., and, written "∧"). A permission is allowed if at least one of the clauses evaluates to true, and disallowed if all clauses evaluate to false. The default policy sets both the permissions to `deny`.

Growlithe defines a set of predicates for arithmetic and string operations, comparisons, and taint tracking. Table 2 shows a subset of the predicates supported in Growlithe's language. (The complete predicate list is in §A.) Specific predicates use arguments that are set or compared against constants or values of the abstract identifiers defined in Table 1. Any taint predicates in the permissions specify conditions to allow data flow from the source to the sink. Taint predicates appear typically in the write permission of the last edge in a workflow, which represents the boundary of the application and requires checks before data is allowed to be exfiltrated.

Taint Labels. Growlithe associates a taint label with data conduits and with functions. For conduits external to functions, e.g., databases, KV stores, queues, Growlithe stores the labels as metadata within the corresponding service. Growlithe supports a distinct label for each object maintained within the service, e.g., each database record, key-value pair, or queue element. For the functions and the conduits internal to functions, e.g., the local file system, the taint labels are stored in the container global memory and as file metadata, respectively.

Predicate	Description
eq(x, y)	Check if x == y.
not(x)	Check if !x is true.
concat(s, x, y)	Assign to s the concatenation of strings x ‖ y.
getVal(v, d, k1, ..., kn)	Assign to v the dictionary value associated with keys k1, ..., kn, i.e., v = d[k1][...][kn].
taintSetIncludes(n, l)	Check if the taint set for ADG node n contains label l.
taintSetExcludes(n, l)	Check if the taint set for ADG node n does not contain label l.

TABLE 2: Subset of Growlithe's policy predicates.

A label for a data conduit is defined as "`ResourceName:ObjName`". The `ResourceName` and `ObjName` may be resolved from the abstract identifiers, Resource and Object, respectively, either statically or at runtime. The label for a *function* is defined as "`InstName`", which is the unique name of the function instance in an application. Typically, the instance name corresponds to the function's static name.

The current label semantics can differentiate multiple invocations of a workflow triggered via direct function calls and via operations on data stores or communication services. The semantics can be easily extended to differentiate the invocations based on other trigger sources, such as user requests (by adding user identification) and periodic events (by adding event timestamps).

The taint predicates can describe a set of labels by under-specifying a part of the label using `*` symbol, indicating that all labels that match on the remaining parts of the label satisfy the predicate. For example, the predicate taintSetExcludes(N, `'r:o'`) specifies that the taint set must not include the label of an object `r.o`, while taintSetIncludes(N, `'r:*'`) specifies that the taint set could include the label of any object in resource `r`.

Policy specification example. The policy **P3** for the ClaimApp in §2.2 can be satisfied by specifying the following policy on edge e_4 in Figure 3:

```
read :- allow
write :- taintSetExcludes(PredNode, 'UserPlan:*')
```

This policy allows writing to n_6 (a sink-only node) if the taint set of the conduit read in the function (node n_4) does not contain any taint label for data from `UserPlan`.

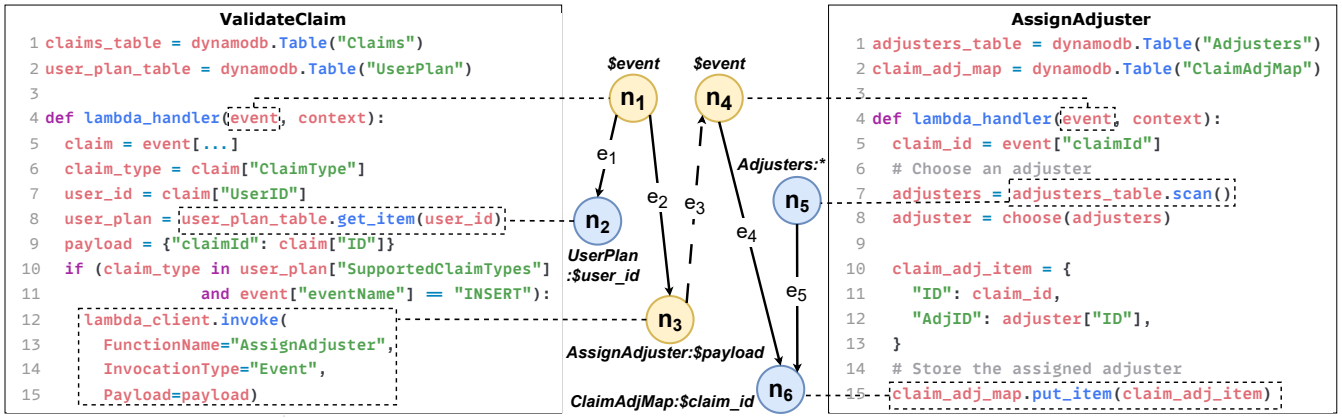

Figure 3: Subset of the Application Dataflow Graph (ADG) for the Claim Processing application, showing the intra-function dataflows (solid edges) and inter-function dataflows (dashed edges) between `ValidateClaim` and `AssignAdjuster`. Blue nodes represent the persisted data stores; yellow nodes represent event source, function response and function payloads.

3.3. Policy Specification Guidelines

When Growlithe is run on an application for the first time, its policy generator configures default template policies for each edge in the application's ADG, which denies all reads and writes. Subsequently, a developer revises the policies on specific edges to allow data access and flows subject to compliance goals. Below we describe the guideline for developers to systematically add policies on the edges of the ADG—starting from "boundary" edges and then moving to the intermediate edges—to progressively allow the minimal set of dataflows required for the application to function.

Policies on edges consist of two types of constraints: *global* and *local*. The developer first identifies any constraints that apply globally on the dataflows between all conduits and functions (e.g., region checks) and prepares the corresponding predicates, which will be added to read policies of edges from source nodes and to write policies of edges to the sink nodes. We now discuss the edge-local constraints and how developers combine them with the global constraints.

1) *Boundary edges*: These are the edges connecting to boundary nodes (i.e., source nodes with no incoming edges and sink nodes with no outgoing edges). The read/write constraints on the boundary edges are of two types. (a) Each source and sink that is statically identified (hardcoded) has a distinct node in the ADG; thus, their read and write policies are added to the individual edges from the source and edges to the sink, respectively. (b) If the sources and sinks are identified as variables in the ADG, meaning they can only be identified dynamically, the policies for each of them are added together, separated with a disjunction, on the edge originating from the source or the edge going to the sink. The effective read permission on the source node of a boundary edge is the conjunction of the global predicates and the read predicates. In contrast, the effective write permission on the sink node of a boundary edge is the conjunction of the global predicates and the write predicates.

2) *Intermediate edges*: Policies may be required on intermediate edges in two scenarios. (a) The intermediate nodes may represent data conduits that are shared across multiple workflows, and different objects from the data conduits need to be isolated across the workflows. (b) The dataflows through intermediate nodes may be subject to checks based on information available only locally at those nodes, e.g., local time or names of files in a local filesystem.

The effective policy on an intermediate edge includes a read permission on the source node that is the conjunction of the global predicates and the read predicates, and a write permission on the sink node that is the conjunction of the global predicates and the write predicates.

3) *Remaining edges*: After all the data access, dataflow, and isolation concerns have been addressed in the previous steps, the developer finally checks if there are any remaining edges with only the default `deny` policies. Each of these edges would be an intermediate edge that is part of only one workflow. Given the conditions placed on the other edges, the leftover edges do not lead to any dataflow violations. Thus, a developer can safely set the read and write permissions on these edges to `allow`, allowing the dataflows and ultimately allowing the application to function.

Revising policies upon application changes. If the application evolves or if policies change, developers may need to re-run the dataflow graph generator and the policy generator to revise the annotated ADG in accordance with the compliance requirements. The policy generator reviews the difference between the current and the previous version of ADG. It drops the policy for each edge dropped from the previous ADG, adds a new default policy for each edge added to the ADG, and preserves the refined policies on existing edges. Subsequently, a developer can revise the policies on all edges in the current ADG appropriately to preserve the compliance guarantees.

3.4. Policy Enforcement

Achieving compliance conceptually entails enforcing the policies specified along the edges of the $\overline{ADG}$. Growlithe enforces policies in two stages. First, during the static analysis stage, Growlithe traverses $\overline{ADG}$ to evaluate policies on all edges. For policies that cannot be evaluated statically, Growlithe adds assertions at appropriate places within the source code of functions. In the second stage, the assertions enforce the deferred policies during function execution. In each stage, enforcing policies entails resolving the arguments in policy predicates (equivalently assertions) and evaluating the predicates (respectively assertions). When a policy evaluation fails, Growlithe logs the error and notifies the developer, who must then revise their application and/or policies. We describe the two mechanisms in detail here.

Static enforcement. Given an $\overline{ADG}$, Growlithe statically checks the policies on the edges. For each edge policy, it tries to check the read permission on the source conduit (respectively, the write permission on the sink conduit) by evaluating the permission clauses and predicates. For predicates referring to abstract identifiers (Table 1), Growlithe attempts to resolve the identifiers with application-specific attributes from various sources. It resolves conduit and API identifiers from $\overline{ADG}$ node properties and function names from configuration files. Predicates relying on time, user session information, or an external service's region cannot be resolved statically and are deferred to runtime checks.

A key challenge is to resolve taint labels for evaluating the predicates taintSetIncludes and taintSetExcludes in any policy. At each ADG node v, Growlithe computes the associated taint label as a union of its current label, the labels of all its predecessor nodes, and the labels of the functions reading from the predecessor nodes and writing to the node's conduit, i.e., $taint(v) =$

$$taint(v) \bigcup_{p \in Pred(v)} taint(p) \bigcup_{p \in Pred(v)} taint(p.ObjFn)$$

where $Pred(v) = \{p : (p, v) \in E\}$ is the set of predecessor nodes of v, and $p.ObjFn$ is the function that reads from p and writes to v.

After resolving the abstract identifiers wherever possible, Growlithe performs a depth-first traversal of the $\overline{ADG}$ and attempts to evaluate the edge policies. For each policy, Growlithe evaluates the read and write permissions separately using a lightweight Datalog logic solver [32]. For each permission, Growlithe may see one of three outcomes during static evaluation. (i) Growlithe may evaluate the permission to `deny`, i.e., all the permission clauses may resolve to `deny`. In this case, Growlithe generates an error indicating the failed policy and the source code location, and alerts the developers. (ii) Growlithe may evaluate a policy to `allow` statically, in which case it eliminates those checks from the application's runtime deployment configuration. (iii) Growlithe cannot fully evaluate a policy, in which case it defers the unresolved predicates to runtime checks.

Source code instrumentation. To support runtime checks, Growlithe modifies the application source code in two ways. First, it transforms the unresolved read (respectively write) permission predicates of an edge into assertion conditions and injects them into the application function (tailored to the function's programming language) before the read (respectively write) operations associated with the edge, respectively. Second, Growlithe initializes a datastructure `GROWLITHE_TAINTS` in each function and adds code to append the labels of the source conduits to it, write to the sink conduits subject to the accumulated taint labels and data-flow conditions, and persist the accumulated taint as metadata in the sink conduit.

Generating deployment configuration. Cloud providers offer IAM tools to regulate type of actions that the functions can perform, and the resources that the functions can access. Even without Growlithe, the IAM role of each function is required when the function is created. Similarly, developers need to set proper IAM roles based on the trigger type of the functions (e.g., over HTTP, by pub/sub messaging, etc.). Ideally, these IAM policies should be configured following the principle of least privilege [33]. However, the complexity of IAM often leads developers to configure overly permissive IAM policies for their applications. These overly permissive policies are often the root cause of unintended data leaks [34].

Using the ADG, Growlithe captures the precise resources accessed—specifically, the cloud data services used for sharing data between different functions of the applications. Additionally, Growlithe captures the actions performed on those resources by the application functions. Growlithe uses this information to auto-generate default IAM policies allowing each function to perform only the actions captured on each of the discovered resources.

IAM policy example. For the claim processing application, for instance, Growlithe generates the following IAM policy allowing the `AssignAdjuster` function the necessary actions on `Adjusters` and `ClaimAdjMap` tables (other resources are truncated due to space constraints):

```
{
  {"Effect": "Allow", "Action": "dynamodb:Scan",
   "Resource": "Adjusters"},
  {"Effect": "Allow", "Action": "dynamodb:PutItem",
   "Resource": "ClaimAdjMap"}
}
```

Runtime enforcement. Since Growlithe's static policy enforcement stage does most of the heavy lifting with respect to enabling compliance, the runtime enforcement mechanism is relatively simple. When a function is executed, the instrumented logic in the function performs the following sequence of operations: (i) for each conduit read by the function, append the conduit's taint label to the function's `GROWLITHE_TAINTS`, (ii) evaluate the policy assertions prior to each conduit read operation, (iii) evaluate the policy assertions prior to each conduit write operation, (iv) complete the conduit write operation, and (v) persist the taint labels accumulated in the function's `GROWLITHE_TAINTS` to the written conduit's taint label. If a policy check fails during runtime, Growlithe aborts the function's execution.

Our prototype focuses on single-threaded functions, which constitute a large fraction of function implementations. In principle, Growlithe can be extended to multi-threaded functions, for instance, by using locks to synchronize assertion checks and data accesses.

3.5. Security Properties

Completeness. Like any static analysis tool, Growlithe requires correct modeling of data conduits for generating an accurate and complete dataflow graph.

Soundness. Assuming conduits are correctly modeled, Growlithe's soundness follows from the soundness of

1) *ADG generation (§3.1):* Growlithe over-approximates dataflows during ADG generation in two ways. (i) It adds edges for all possible data flows to a sink node, regardless of the control flow. (ii) For applications involving black-box third-party libraries, it assumes all possible dataflow paths between the library interface and its predecessor and successor functions. This ensures that no dataflows are missed. (False positives are overcome by runtime checks.)

2) *Policy specification (§3.3):* Policies are specified according to the least privilege principle. Initially all policies are configured to deny dataflows. Each step allows a developer to carefully reason about conditions to allow dataflows and explicitly configure those conditions as edge policies. Furthermore, the effective policy enforced on a workflow is the conjunction of the read and write permissions along all the edges in the path, which reduces chances of inadvertent dataflows due to overly permissive policies at any edge.

3) *Static enforcement (§3.4):* Predicates that cannot be checked statically are added as assertions in the functions for runtime checking. In particular, Growlithe evaluates taint predicates by analyzing all the path prefixes of a sink node: if any path fails to satisfy the predicate, Growlithe defers the taint check to runtime. Thus, no predicates are missed.

4) *Policy generation (§3.4):* Growlithe's assertion and IAM policy generator is a part of its trusted computing base. The IAM policy generator configures coarse-grained access control only between functions and resources observed during the static analysis and follows the least privilege principle; other functions and resources in the cloud account are denied access by default. The effective policy at an ADG node is the combination of the IAM policy and assertions.

5) *Runtime enforcement (§3.4):* Soundness is guaranteed from our threat model, since the application and the cloud platform do not bypass runtime checks.

4. Implementation

We have implemented a full prototype of Growlithe[2]. We build Growlithe's offline-stage components on top of CodeQL [29], an industry-grade code analysis tool. CodeQL supports multiple programming languages and provides several intermediate representations (IRs) that Growlithe builds upon. Growlithe integrates its runtime components directly in the application code that is deployed on a cloud platform.

Growlithe consists of two modules: the Core module written in Python and the Program Analysis module using CodeQL. First, the Program Analysis module parses each function's source code and generates its data dependency graph by leveraging the model for data conduits, conduit APIs, and the rules for data flow through them. It annotates the source and sink nodes within each function's IR and then issues queries to generate a SARIF file [35]. Next, the Core module takes the SARIF input and generates the function's dataflow graph. It then parses the application configuration files, e.g., orchestration configuration, which provide information about the application's control flow across the functions and the resources used in the application. Growlithe combines this information with the per-function dataflow graphs to generate the ADG.

Using inputs from Growlithe and following the guidelines from §3.3, developers specify policies on each edge of the ADG in the form of a JSON file. Subsequently, Growlithe relies on pyDatalog [32] for evaluating the policies both during static and runtime enforcement. During static enforcement, Growlithe performs a breadth-first traversal of the ADG and evaluates the policy associated with each edge. For policies deferred to runtime checks, Growlithe adds assertions ahead of the read or write operations on data conduits in the source code of appropriate functions. It also generates the IAM policies and adds them to the application's deployment configuration.

Growlithe's Core and Program Analysis modules are implemented in 1500 and 2000 LoCs, respectively. Growlithe could be deployed as a plugin integrated with client-side IDEs (integrated development environments) [36, 37], or with server-side SDKs [38, 39] or cloud tools [33, 40].

Growlithe currently supports Python and JavaScript functions. In §5.1-§5.3, we focus on evaluations with applications built for AWS Lambda [41], one of the leading serverless platforms [20, 42]. In §5.4, we discuss the costs for supporting additional languages and platforms.

Growlithe only requires manual input from the developers for the policy specification on the ADG edges. Given the specification, Growlithe automates the end-to-end pipeline for enforcing the policies on serverless applications even as the application evolves.

5. Evaluation

Through our evaluation, we assess (i) Growlithe's generality in enforcing policies in diverse applications (§5.1), (ii) the costs of Growlithe's pre-deployment steps and runtime enforcement at scale (§5.2), (iii) the impact of Growlithe's costs on real-world applications (§5.3), (iv) the cost for extending Growlithe to support additional services and languages (§5.4), and (v) Growlithe's ability to detect and prevent policy violations (§5.5).

We ran Growlithe's pre-deployment steps on an Apple M1 MacBook Pro (8 cores, 16 GB RAM, MacOS 14.4.1),

2. https://github.com/ubc-cirrus-lab/growlithe

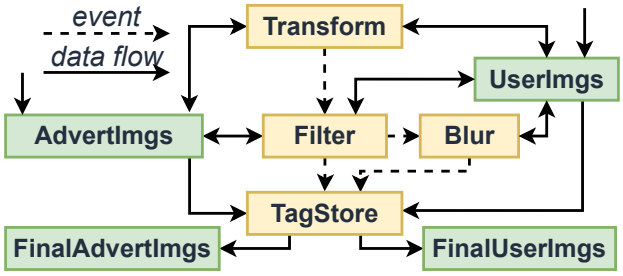

Figure 4: Image Processing application.

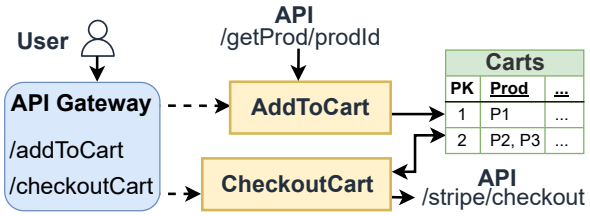

Figure 5: Shopping Cart application.

where we generate an application's ADG, policies, and the instrumented code for deployment.

We evaluated Growlithe's runtime enforcement on the AWS Lambda platform. We deployed all functions of each microbenchmark (§5.2) or application (§5.3) in the ca-west-1 region within single containers configured with 1769 MB of RAM, and we report the results from runs on Intel Xeon 2.90 GHz CPUs. The container configuration provides one full vCPU [43], which eliminates the effects of CPU throttling in our measurements and is, therefore performance-optimal for our single-threaded benchmarks. This allows us to assess the true performance overheads of Growlithe.

5.1. Case Studies

We demonstrate Growlithe's generality by applying it on three realistic serverless applications that can run on AWS Lambda. These applications cover different characteristics, such as diverse data stores, inter-function communication mechanisms, programming languages, and data policies. Furthermore, the number of functions, as well as the width and depth of the DAGs (directed acyclic graphs) in the applications reflect the 95th percentile complexity of real-world serverless applications [44].

Claim Processing (CP). The application and the required policies are described in §2.2 and Figure 1.

Image Processing (IP). Figure 4 shows an image processing application, which retrieves raw images from an AWS S3 bucket, pre-processes the images through a series of functions that are orchestrated via AWS Step Function, and stores the images in another S3 bucket. We developed the application by extending the image processing benchmark [45] into two workflows. We consider the scenario of a social media service using this application to store and process images from different sources. Specifically, the application handles both users' private photos and advertisers' privacy-insensitive images.

The raw images from users and advertisers are stored in two separate input buckets `UserImgs` and `AdvertImgs`, respectively (`raw/` directories in each bucket), and the final processed images are stored in output buckets `FinalUserImgs` and `FinalAdvertImgs`, respectively. The AWS Step Function orchestrates the sequence of processing functions `Transform`, `Filter`, `Blur`, and `TagStore`, which operate on one image at a time. The first three functions each download an image file from one of the input buckets into their local file system ("`/tmp/<inputimg>.<fnname>`"), process the image locally, and then upload their output image as a separate file back to the same bucket. The final function `TagStore` reads the input image from an input bucket and stores the output image into the corresponding output bucket. The application is triggered when an image is uploaded to either input buckets, which is passed to the orchestrator. The orchestrator passes the output image of one function as the input for the next function. Each function uses their local file system as a temporary cache while processing their image.

A key highlight of the application is that it involves an input-dependent control flow. Users' photos need to be subject to all three processing functions, while the `Blur` function is not invoked when processing advertisers' images.

The social media service wishes to (i) restrict processing of images within the same geographical region where the images are stored, (ii) adhere to users' privacy preference of blurring the background in their images, and (iii) prevent users' images from being shared with advertisers. These requirements translate into the following policies:

P4 Process images in the region that they are stored. The region of all functions and data stores used in the application should be the same as the original bucket where the raw images were stored.

P5 Blur user images. Images stored in `FinalUserImgs` must have passed through the `Blur` function.

P6 Prevent sharing of user images with advertisers. Images from `UserImgs` should not be written to `FinalAdvertImgs`.

P7 Restrict cloud store for intermediate images. The intermediate functions (`Transform`, `Filter`, `Blur`) should store their outputs in the bucket from which they read the input images to process.

Shopping Cart (SC). Figure 5 shows a part of a shopping cart application [46], applied to an e-commerce scenario. The application maintains a per-user "cart" record in a `Carts` table. The `AddCart` function handles customers' requests to add products to their cart, while `CheckoutCart` allows customers to purchase the products in their cart and make payment. The `CheckoutCart` function invokes the public API of a third-party payment service to handle payment. We retain `CheckoutCart` in Python but re-implemented `AddCart` in JavaScript. This demonstrates

Application	W	Fns	Tmplt	Pols	Preds	Runt
Claim Processing	4	5	9	3	13	12
Image Processing	1	4	11	8	19	10
Shopping Cart	2	2	6	4	11	8

TABLE 3: Policy specification statistics. (W = workflows, Fns = functions, Tmplt = template policies, Pols = policies, Preds = specified predicates, Runt = runtime predicates.)

Growlithe's usability in multi-language applications.

The e-commerce company operates globally. For compliance reasons, the company may be required to process customers' payments using payment gateway services that operate within the customers' region. Moreover, the company may need to ensure that the functions operate on the customers' cart in the same region as the customer as well. This would translate to the following policies

P8 **Payment region restriction:** `CheckoutCart` should ensure that the region of the payment API matches with the region of customer's request.

P9 **Checkout region restriction:** The region of the two functions, the `Cart` record, and the customer's requests should match with each other.

Modeling costs. The SC, CP, and IP applications consist of 493, 575, and 243 LoCs, respectively. We modeled 17 and 3 APIs in CodeQL for Python and JavaScript, respectively, which required 450 and 110 LoCs, respectively. We deploy these applications using the AWS Serverless Application Model (SAM) [40]. We added 550 lines to parse the SAM configuration and to automatically add IAM policies. Lastly, we implemented the taint infrastructure in 390, and the translators for policy predicates and identifiers in 180 lines. We present the ADG and policies of all applications in §B.

Policy statistics. In Table 3, columns 2, 3, and 4 respectively indicate the number of distinct workflows in the application, the total number of functions in the application across all workflows, and the number of ADG edges shown in the policy template generated for each application. As mentioned in §3.3, each of these edges is assigned a `deny` policy initially, and developers revise the policies to enable data flows in the application. For each application, column 5 indicates the number of edges in the template, where developers need to specify a non-trivial policy; the remainder of the edges from column 4 are configured with an `allow` policy. Column 6 indicates the total number of predicates across all the non-trivial policies: each such policy contains 4 to 7 predicates between the read and write permissions. Of these predicates, column 7 indicates the number of predicates that are checked at runtime. Thus, Growlithe could statically check 7-47% of the policy predicates in these applications.

In general, Growlithe can statically check for the names and properties of conduits and functions (e.g., the region of a deployed bucket) and dataflow paths (e.g., images passing through a specific function), which helps in debugging coarse-grained workflow-level dataflows. The runtime checks complement static checks with fine-grained user- or invocation-specific policy enforcement.

5.2. Microbenchmarks

We ran microbenchmarks to break down the costs of Growlithe's pre-deployment steps and runtime enforcement, and to assess the scaling of the costs with larger applications.

We created synthetic workflows with the number of functions varying from 1 to 128, and with the functions either chained linearly (**LC**) or invoked concurrently in a fanout (**FO**) configuration. All the functions are similar: they retrieve a file from an S3 bucket, wait for 65ms (around the median execution time of a serverless function [20]), and subsequently write the file back to the same S3 bucket. In each workflow configuration, we configure a uniform policy on all edges with one of three predicate types: (i) **remote**, which checks that the region of the S3 bucket (must be fetched over the network at runtime) matches that of the function(s) accessing it (available local to the function at runtime), (ii) **local**, which performs similar checks as remote, except that the region of the S3 bucket is statically configured, thus avoiding network requests, and (iii) **taint**, which checks that each function's taint set contains preceding function(s) before allowing write to its output bucket.

Pre-deployment costs. We split the pre-deployment costs into two components: (i) *analysis*, which includes the time required to generate CodeQL IR and the ADG from it, and (ii) *apply*, which includes the time required to perform static policy enforcement, add policy assertions in functions, and create the deployment configurations with IAM rules. (We ignore the costs for policy specification, as they involve human intervention and are highly variable.)

We ran the pre-deployment steps for each workflow and policy configuration 10 times. The pre-deployment cost is independent of the application size and configuration. The average cost observed across all experiments was ~43.37 s, of which, *analysis* constitutes ~43.25 s. (Standard deviations are <10%). In principle, the IR and ADG can be built incrementally from prior builds in subsequent iterations [47]. We defer these optimizations to future work.

This shows that Growlithe adds low overhead in the pre-deployment steps, making it easy for developers to adopt Growlithe in their development process.

Runtime costs. Figure 6 shows the end-to-end execution times of **LC** and **FO** workflows (left and right plots, respectively) in the baseline and with the three policy configurations. We run each configuration 50 times and apply the Interquartile Range (IQR) method [48] to exclude cold-start times by removing the outliers above $Q_3 + 1.5 \times IQR$, where Q_1 and Q_3 are the first and third quartiles, respectively. We report the mean over the remaining measurements, with error bars showing standard deviation.

The average per-function cost is negligible for a local policy check, while it is 11.7 ms and 24.7 ms for taint and remote policy checks, respectively, in **LC** workflows and 13.4 ms and 19.6 ms, respectively, in **FO** workflows. In practice, policies may include a mix of predicates, some

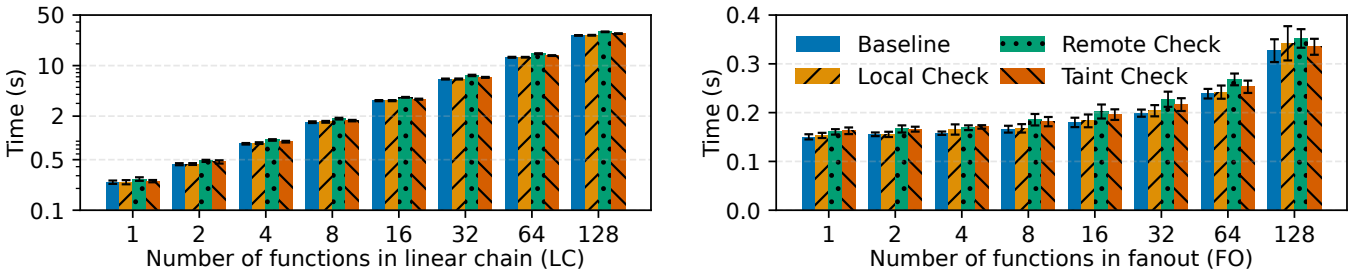

Figure 6: Runtime performance for microbenchmarks.

of which may even be enforceable statically. Thus, the high overheads in a single function do not necessarily translate to high end-to-end overheads for entire workflows. We discuss this further in the next section.

5.3. Performance Costs on Real Applications

Next, we evaluate Growlithe's pre-deployment and runtime costs on the real applications from §5.1.

Pre-deployment costs. Similar to the microbenchmarks, the pre-deployment costs were ~49 s for CP and IP, and ~62 s for SC. The higher cost for SC is due to having to run *analysis* steps twice (for Python and JavaScript code), which currently execute sequentially. The costs can be lowered by parallelizing the steps across languages.

Runtime costs. We compare the runtime costs of three configurations: (i) **Base**, which corresponds to running an insecure application in vanilla AWS Lambda containers with default IAM policies configured manually to allow application execution, (ii) **Growlithe**$_{RT}$, which corresponds to running an application with runtime policy enforcement alone, i.e., all its policies are enforced with only runtime assertions, and (iii) **Growlithe**$_{Opt}$, which corresponds to running an application after applying static enforcement, which potentially leads to fewer checks during runtime enforcement. We measure the code size of the functions after instrumentation, the runtime memory utilization of the individual functions, the execution latency of individual functions, the cold start latency of functions including container boot up times, and the end-to-end latencies of the complete workflows. For the latency metrics, we collect 100 measurements using the same input per benchmark for each experiment, and apply the same pruning as in §5.2 to account for the serverless platform's execution variability [21].

Function code size. The code size of the functions in the **Base** configuration ranges from 0.5 KB to 8 KB (median: 0.6 KB). The overhead on the code size is highest with **Growlithe**$_{RT}$ configuration, where all policies are added to the code as assertions. Therefore, we only compare with the code size of the functions in this configuration, which ranges from 1.6 KB to 8 KB (median: 2 KB). The maximum overhead on the code size across all functions is 3.1 KB.

Memory utilization. Memory overhead is a critical metric because an application requiring more memory would require a larger and more expensive container. Growlithe's max memory overhead (which is due to code assertions and the taint tracking infrastructure integrated with each function) is 9 MB across our applications. The highest overheads are for **Growlithe**$_{RT}$, which does not have any static enforcement that could reduce the assertions in the code and, therefore, reduce the container memory size.

Function cold start latency. Growlithe's library and code instrumentation increase the cold start latency of a function, i.e., the time a request has to wait before the function is available to process the request. Figure 7A shows the average cold start latency for the three configurations. The maximum cold start overhead is ~178 ms, which is incurred in the IP application. The high overhead is because we move some costs of runtime enforcement to the container initialization phase. Specifically, for the predicate checks that involve looking up remote resources, we initialize connections to those resources during container initialization. For predicates that can be checked locally, the cold start overhead for functions is ~14 ms on average.

Function execution latency. Figure 7B shows the average function execution time for the three configurations. The error bars show the standard deviation. **Growlithe**$_{RT}$ and **Growlithe**$_{Opt}$ configurations incur an average overhead of 28 ms and 23 ms over the **Base** configuration, respectively. Most of the runtime checks are lightweight and incur negligible overheads, but the maximum overheads for **Growlithe**$_{RT}$ and **Growlithe**$_{Opt}$ are 102 ms and 98 ms, respectively. These overheads are due to the region-check policy assertions, which require fetching the region information from external sources and thus incur network latency costs. The **Growlithe**$_{Opt}$ configuration is on average 19% faster than **Growlithe**$_{RT}$. The reduction in runtime overheads is proportional to the number of predicates that are checked and eliminated statically (Table 3).

The remaining overheads on individual functions due to remote policy checks could be mitigated by caching the results of external requests involved in policy checks.

Workflow execution time. We measure the workflow execution time as the time between the start of the first function execution in the workflow upon receiving an event and the completion time of the last function. Using Growlithe, for single-function workflows, the end-to-end overheads closely resemble the overheads of individual functions. The

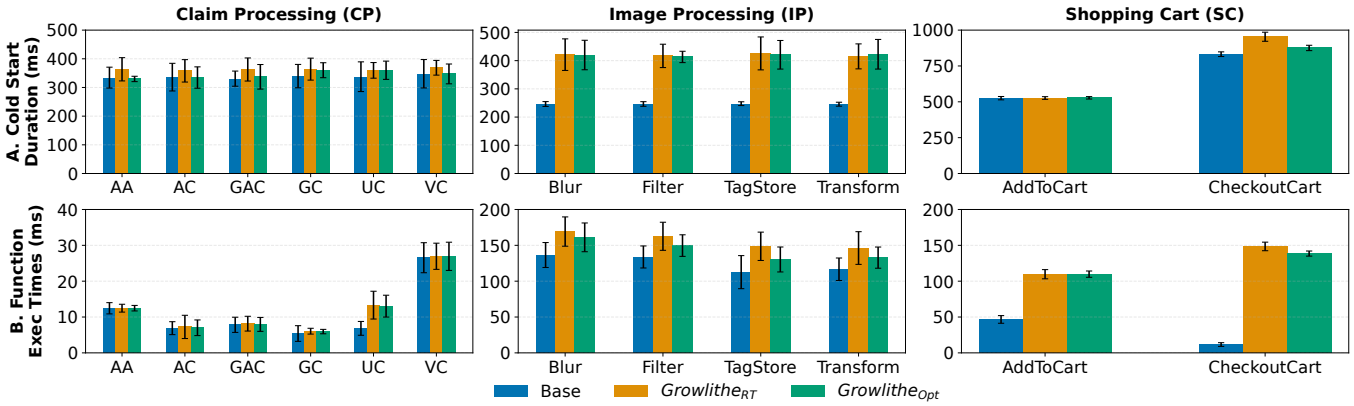

Figure 7: Cold start and execution times of Growlithe vs. Base. For Claim Processing, the function names are as follows. AC: AddClaim, AA: AssignAdjuster, GAC: GetAssignedClaims, GC: GetClaim, UC: UpdateClaim, VC: ValidateClaim.

absolute overheads are in 1-102 ms range, which are negligible from an end user's perspective. In contrast, for more complex workflows, consisting of multiple functions and resources, the relative end-to-end overheads are significantly lower (e.g., only an overhead of 9.5% and 1.7% for IP and CP's AddClaim workflows, respectively).

Summary. Static enforcement helps reduce runtime performance overheads. These reductions accrue over a large number of function invocations, resulting in cost savings for the application. Overall, we demonstrate that Growlithe provides a practical compliance tool for serverless applications.

5.4. Porting Costs

We ported the ShoppingCart (SC) application to the Google Cloud Platform (GCP), deployed it using a Terraform [49] template and added the necessary support for GCP in Growlithe. We added 318 LoC in CodeQL to support 13 datastore APIs, 60 LoC in Python to implement the required policy predicates, 247 LoC in Python to parse the Terraform deployment template for GCP and automatically generate IAM policies. This effort took 6 person-hours.

In general, the costs of supporting multiple languages and services on a cloud platform are as follows. Assuming D conduits (cloud services), N dataflow APIs and L language implementations for each conduit, the ADG generation effort requires modeling $O(D{*}N{*}L)$ dataflow rules. Furthermore, Growlithe's enforcement requires a translator per language to generate assertions from policy predicates and to retrieve values for abstract identifiers in the predicates from the cloud platform. Additionally, a developer needs to handle each cloud's semantics to generate the IAM policies.

Note that all of these are one-time costs for each conduit and language to be supported. The tools and models could be maintained by the community, similar to CodeQL [29, 50].

5.5. Security and Usability Evaluation

We discuss three scenarios, showing (i) Growlithe can detect bugs in an existing source code violating a data policy **(S1)**, (ii) the effort required to revise a policy specification upon changing an application workflow **(S2)**, and (iii) the effort required to revise an application upon a change in policy requirement **(S3)**.

S1. Inadvertent sharing of data. In the CP application, suppose a developer revises an automated check for claim validation in the `ValidateClaim` function, which reads some attributes of a `UserPlan` record. Suppose, the developer accidentally passes some part of this record, which is a numeric value (e.g., a user's social security number), to the `AssignAdjuster` function instead of the claim id, another numeric value. In absence of Growlithe, `AssignAdjuster` could insert the user's social security number in the table, revealing it to the assigned adjuster and violating the user's privacy. Fortunately, the policy **P3** on edge e_4 (Figure 3) would detect the bug during static enforcement and notify the developer.

S2. Adapting to a new policy. The *i*Claim company revises their privacy policy to state that each customer claim is processed by only one adjuster. A developer could enforce this policy in the CP application with a single policy update on edge e_5 (Figure 3) as follows (read is allowed):

$$\begin{aligned}\text{write :- }\ & \text{getKey}(\text{C}, \texttt{'event'}, \texttt{'claimId'})\\ & \wedge\ \text{concat}(\text{L}, \texttt{'ClaimAdjMap:'}, \text{C})\\ & \wedge\ \text{taintSetExcludes}(\text{CurrNode}, \text{L})\end{aligned}$$

The policy allows `AssignAdjuster` to write an entry to `ClaimAdjMap` for the claim id, `C = event['claimId']`, if the function is not already tainted with the label for `ClaimAdjMap` record of the same id.

Once the policy is updated, the application would trigger an assertion failure due to policy violation at runtime. A user's update to her claim record would trigger the validation pipeline, in which the `AssignAdjuster` function would randomly select a new adjuster, and attempt to overwrite the previously assigned adjuster. To fix the bug, the developer would also need to revise the `AssignAdjuster` implementation to perform a similar check prior to inserting a record in the `ClaimAdjMap` table.

```
1  def lambda_handler(event, context):
2      # pre processing
3      obj = event['ObjKey']
4      tmpfile = '/tmp/' + obj
5      if (not os.path.exists(tmpfile)):
6          bucket.download_file(obj, tmpfile)
7      # ...process tmpfile and upload obj
```

Figure 8: `Blur` function with a container reuse vulnerability.

S3. Retaining compliance when adding new functionality. Recall from the IP application, that the first three functions in the workflow download an input image into their local file system at a path identified by "`/tmp/<inputimg>.<fnname>`". As shown in line 4 of Figure 8, the application initially configured the `Blur` function to download an image, say "`UserImgs/photobooth.filter`" to "`/tmp/photobooth.blur`", i.e., the local file name was derived only from the input file's name in the S3 bucket and excluded the bucket's name.

Now suppose, the social media service provider wishes to post images from its product launch event for publicity, but wishes to ensure that no information about unreleased products is visible in the posted images. A developer could quickly accomplish the task by simply replicating the IP workflow that is used for users' images and changing the input and output buckets to, say `EventImgs` and `FinalEventImgs`, respectively.

In the absence of Growlithe, there is a potential leak of users' images to the new events workflow via the local file system of the `Blur` function (lines 6 in Figure 8). If a `Blur` instance cached a user image in the local file system and was reused to process an event image with the same file name, the function may process the cached image instead of downloading the image from the `EventImgs` bucket. With Growlithe, however, the policy **P7** will detect this inadvertent leak at runtime and alert the developer.

6. Discussion

Completeness of dataflow coverage. Growlithe's efficacy depends on the coverage of dataflows in the application (§3.5). While the current prototype models dataflows through cloud services, it does not cover dataflows in two scenarios. First, it does not cover dataflows through callbacks, closures, and some I/O libraries (e.g., csv, pandas for Python; fs-extra, fast-csv for Javascript). However, Growlithe can be easily extended to cover these.

Secondly, the prototype does not cover reflective programming constructs, where an application generates new data or code at runtime. Such constructs did not appear in the applications we studied. Nevertheless, Growlithe can be extended to handle dynamically generated data by analyzing the generator code for the application. Growlithe's runtime enforcement can automatically handle new control flows that do no bypass instrumentation, whereas control flows that do bypass instrumentation are considered malicious behavior and, hence, are outside the scope of our threat model.

A practical challenge for Growlithe is to keep up with the evolutions in the programming languages. Growlithe must be updated to cover any new control and data flow scenarios arising from newer constructs introduced in future language versions. Large language models could help in automatically modeling new conduits and dataflows [51].

Ensuring policy correctness. Growlithe's language-independent ADG and declarative policy language are the first steps towards reducing policy specification errors for serverless. Policy specification could be further simplified by automatically generating policies in Growlithe's language from natural language specifications [52, 53]. Additionally, Growlithe could be enhanced with tools to visualize the annotated ADG and for testing and debugging to enable reasoning about the correctness of policies.

7. Related Work

Compliance solution architectures. With the enactment of modern data privacy regulations, several compliance solutions have been proposed for software systems. Some solutions integrate compliance mechanisms within the data layer [54–57], but focus on a single type of data store: block storage or a database management system. Purview [58] centralizes governance of data across multiple structured data stores and enables enforcing fine-grained access control. These solutions are insufficient to ensure compliant data use in applications where data is processed across multiple components backed by heterogeneous service stacks.

Other solutions integrate policy enforcement within web services, e.g., at the interface between the application logic and the database adapter [17], within language runtime [19], or using static analysis on application source code [18]. These solutions were designed for conventional web applications, which are different from serverless applications that Growlithe supports, and cannot adapt to application changes.

RuleKeeper [59] enables compliant use of users' personal data based on their consent in MVC-style web frameworks. RuleKeeper enforces access control policies in each controller function through static analysis and runtime enforcement. Growlithe enables compliance in serverless applications, which are designed as workflows consisting of several functions communicating directly or through heterogeneous datastore or communication services. Like RuleKeeper, Growlithe also relies on modeling data conduits and enforces policies through a hybrid static-dynamic mechanism. Unlike RuleKeeper, however, Growlithe can enforce policies on end-to-end dataflows within a serverless application. Moreover, instead of relying on a middleware for policy enforcement, which requires platform support, Growlithe enforces policies through assertions in application code.

Security in serverless applications. Trapeze [24] enforces dynamic and fine-grained IFC in serverless applications, which requires modifications to data stores and the function's language runtime. It protects against a strong

threat model where functions are mutually distrusting and data can leak through storage and implicit termination channels. Growlithe uses a hybrid mechanism to enforce fine-grained IFC in serverless applications, which is more efficient, and is independent of the function's runtime. Growlithe's IFC model could also be enhanced to provide termination-sensitive noninterference guarantees.

Valve [22] enforces coarse-grained dataflow policies, controlling the data services that should be accessible by specific functions in different workflows. Applications need to use a custom container, which integrates a network proxy to discover data flows between functions, data store and communication services at runtime, and a runtime monitor to perform taint tracking at the boundaries of network I/O and filesystem calls. Growlithe focuses on enforcing fine-grained policies addressing the data privacy requirements of end users as well as the application. Furthermore, Growlithe operates in the application layer, independent of the cloud platform and tools. Finally, Growlithe's hybrid enforcement mechanism enables a more agile application development lifecycle: Growlithe can adapt to application changes by re-running the static analysis in the offline stage and re-deploying the functions only if the instrumentation changes.

Watchtower [60] models checks, such as whether a user has consented to the use of data accessed by functions, or whether sensitive data accesses are logged for auditing, as safety properties, and detects violations of these properties in a serverless application through runtime monitoring. Growlithe can directly specify such policies using its policy specification language, and enforce the policies at runtime.

Cloud security tools. Most cloud providers offer an Identity and Access Management (IAM) tool [8–10, 61] for specifying access control policies for an application and its resources. The tools vary in their levels of support for complex and fine-grained access control policies. Cloud providers also provide a portfolio of tools for orthogonal purposes, such as resource configuration management [62] and software vulnerability detection [63, 64]. None of these tools support dataflow graph generation and analysis by application developers for data policy compliance.

Policy specifications. Prior work has extensively explored declarative policy languages for trust management [65–67] and data policy enforcement [16]. Similar to these efforts, Growlithe also uses a declarative Datalog-based specification language. Resin [13] embeds developer-specified policies as assertions within the application source code and annotates data variables for policy enforcement. By allowing declarative policy specification on the application's ADG and then automatically inserting assertions derived from these policies into the source code, Growlithe reduces developer effort for policy specification compared to Resin.

Dataflow analysis tools. Dataflow tracking and analysis is central to many types of applications, such as fault localization or root cause analysis for distributed software [68, 69], and automatic causal inference between events [70, 71]. These tools typically track dataflows at the network layer and do not scale to application workloads consisting of concurrent or inter-leaved requests between its components, such as those arising in serverless applications.

Alternatively, dataflow analysis can be done per application through static or dynamic analysis [72]. Existing tools for assessing security and privacy vulnerabilities [73, 74] can only support applications in one language at a time and do not cover I/O. Hence, they are not suitable for end-to-end dataflow analysis required for serverless applications.

GRASP [75] creates reachability graphs in a serverless application from existing IAM policies and resource configurations, which developers can query to determine if any application resources are inadvertently publicly exposed. AUTOARMOR [23] discovers communications between microservices through static analysis and automatically specifies access control policies on the microservices. POLYCRUISE [76] is a dynamic flow analysis tool to discover data leaks and code vulnerabilities in multi-lingual programs (covering Python and C). It suffers from false negatives, since it can only discover vulnerabilities in the parts of a program actually executed at runtime, and from high runtime overheads, making it unsuitable for serverless applications. CtxTainter [77] uses dynamic dataflow analysis to only detect leaks across requests in serverless applications.

In contrast to these tools, Growlithe enforces fine-grained access and flow control policies across multi-lingual serverless functions, and combines static and runtime enforcement to achieve coverage and low performance costs.

8. Conclusion

To the best of our knowledge, Growlithe is the first cross-service and cross-language serverless compliance tool that works without requiring changes in the underlying infrastructure, enabling easy adoption. Growlithe builds a fine-grained application dataflow graph, allows developers to specify their policies using a rich Datalog-style syntax, and enforces the policies through a hybrid static-dynamic mechanism. We show that Growlithe can be used for securing serverless applications in practice with low overhead.

Growlithe takes the first step to empower developers to meet their application-specific compliance goals as part of the serverless development lifecycle. Achieving compliance can be further eased with the help of recent advancements in language models, which we leave to future work.

Acknowledgements

We thank the reviewers and our shepherd for their constructive feedback. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) [DGDND-2021-02961, GPIN-2021-03714, DGECR-202100462], the funding from the Innovation for Defence Excellence and Security (IDEaS) Program of the Department of National Defense [MN3-011], the UBC STAIR (Support for Teams to Advance Interdisciplinary Research) Program, and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

References

[1] T. Shea, "Shared Responsibility Model in Cloud," https://sonraisecurity.com/blog/the-shared-responsibility-model-in-the-cloud/, 2023.

[2] "AWS Shared Responsibility Model," https://aws.amazon.com/compliance/shared-responsibility-model/.

[3] "Shared responsibilities and shared fate on Google Cloud," https://cloud.google.com/architecture/framework/security/shared-responsibility-shared-fate.

[4] "Shared responsibility in Cloud," https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility.

[5] European Parliament and Council of the European Union. Regulation (eu) 2016/679 of the european parliament and of the council. [Online]. Available: https://data.europa.eu/eli/reg/2016/679/oj

[6] California State Legislature. California consumer privacy act of 2018. [Online]. Available: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375

[7] Govt. of Canada. Consumer Privacy Protection Act. [Online]. Available: https://ised-isde.canada.ca/site/innovation-better-canada/en/consumer-privacy-protection-act

[8] "AWS Identity and Access Management," https://aws.amazon.com/iam/.

[9] "Azure Identity and Access Management," https://azure.microsoft.com/en-ca/products/category/identity.

[10] "Google Cloud Identity and Access Management," https://cloud.google.com/security/products/iam.

[11] "AWS VPC Network," https://docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html.

[12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information Flow Control for Standard OS Abstractions," *ACM SOSP*, 2007.

[13] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving Application Security with Data Flow Assertions," *ACM SOSP*, 2009.

[14] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, "Hails: Protecting Data Privacy in Untrusted Web Applications," *USENIX OSDI*, 2012.

[15] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing, "Bootstrapping Privacy Compliance in Big Data Systems," *IEEE S&P*, 2014.

[16] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel, "Thoth: Comprehensive policy compliance in data retrieval systems," *USENIX Security*, 2016.

[17] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel, "Qapla: Policy compliance for database-backed systems," *USENIX Security*, 2017.

[18] L. Wang, J. P. Near, N. Somani, P. Gao, A. Low, D. Dao, and D. Song, "Data Capsule: A New Paradigm for Automatic Compliance with Data Privacy Regulations," *VLDB Workshop Poly*, 2019.

[19] F. Wang, R. Ko, and J. Mickens, "Riverbed: Enforcing User-Defined Privacy Constraints in Distributed Web Services," *USENIX NSDI*, 2019.

[20] Datadog, "The State of Serverless," https://www.datadoghq.com/state-of-serverless/, Accessed: Mar 26, 2024.

[21] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," *USENIX ATC*, 2020.

[22] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates, "Valve: Securing Function Workflows on Serverless Computing Platforms," *ACM WWW*, 2020.

[23] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic Policy Generation for Inter-Service Access Control of Microservices," *USENIX Security*, 2021.

[24] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein, "Secure Serverless Computing Using Dynamic Information Flow Control," *ACM OOPSLA*, 2018.

[25] "AWS Step Functions," https://aws.amazon.com/step-functions/.

[26] M. Tom, "Branch's Serverless First Architecture Approach Makes it Easy to Bundle Home and Auto Coverage," https://aws.amazon.com/blogs/startups/branch-insurance-makes-it-easy-to-bundle-home-and-auto-coverage/, 2020.

[27] "Stroll: how we built a complete car insurance platform in 12 months," https://instil.co/case-studies/stroll-insurance/, Accessed: May 28, 2024.

[28] "Proving Serverless for Insurance | Addresscloud," https://www.addresscloud.com/blog/proving-serverless-for-insurance, Accessed: May 28, 2024.

[29] "CodeQL," https://codeql.github.com/.

[30] W. Masri, "Automated fault localization: advances and challenges," *Advances in Computers*, vol. 99, 2015.

[31] N. Li and J. C. Mitchell, "Datalog with constraints: A Foundation for Trust Management Languages," *PADL*, 2003.

[32] "pyDatalog," https://github.com/pcarbonn/pyDatalog.

[33] "AWS Access Analyzer," https://aws.amazon.com/iam/access-analyzer/.

[34] N. Eddy (Dark Reading), "Cloud Misconfig Exposes 3TB of Sensitive Airport Data in Amazon S3 Bucket: 'Lives at Stake'," https://www.darkreading.com/application-security/cloud-misconfig-exposes-3tb-sensitive-airport-data-amazon-s3-bucket, 2022.

[35] "Static Analysis Results Interchange Format (SARIF) Version 2.1.0," https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

[36] "Visual Studio," https://visualstudio.microsoft.com/.

[37] "Eclipse IDE," https://eclipseide.org/.

[38] "Google Cloud SDK," https://cloud.google.com/sdk.

[39] "Azure SDK," https://github.com/Azure/azure-sdk.

[40] "AWS Serverless Application Model," https://aws.amazon.com/serverless/sam/.

[41] J. Bar, "AWS Lambda – Run Code in the Cloud," https://aws.amazon.com/blogs/aws/run-code-cloud/, 2014.

[42] F. Richter, "Amazon Maintains Cloud Lead as Microsoft Edges Closer," https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/, Accessed: May 07, 2024.

[43] "Configure Lambda function memory," https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html, Accessed: Oct 04, 2024.

[44] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows," *ACM POMACS*, vol. 6, no. 2, 2022.

[45] G. Sadeghian, M. Elsakhawy, M. Shahrad, J. Hattori, and M. Shahrad, "UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms," *USENIX ATC*, 2023.

[46] "Serverless Shopping Cart Microservice," https://github.com/aws-samples/aws-serverless-shopping-cart, commit id: 66a863f.

[47] T. Szabó, "Incrementalizing Production CodeQL Analyses," *ACM ESEC/FSE*, 2023.

[48] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions," *EuroSys*, 2023.

[49] "Hashicorp terraform," https://www.terraform.io/.

[50] "CodeQL Supported languages and frameworks," https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/.

[51] W. Chabbott and F. Coada, "CodeQL team uses AI to power vulnerability detection in code," https://github.blog/2023-09-12-codeql-team-uses-ai-to-power-vulnerability-detection-in-code, Accessed: Apr 23, 2024.

[52] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: Interactively translating unstructured natural language to temporal logics with large language models," *CAV*, 2023.

[53] C. Hahn, F. Schmitt, J. J. Tillman, N. Metzger, J. Siber, and B. Finkbeiner, "Formal specifications from natural language," *arXiv preprint arXiv:2206.01962*, 2022.

[54] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post, "Guardat: Enforcing data policies at the storage layer," *EuroSys*, 2015.

[55] T. Kraska, M. Stonebraker, M. Brodie, S. Servan-Schreiber, and D. Weitzner, "SchengenDB: A Data Protection Database Proposal," *VLDB Workshop Poly*, 2019.

[56] A. Agarwal, M. George, A. Jeyaraj, and M. Schwarzkopf, "Retrofitting GDPR Compliance onto Legacy Databases," *VLDB*, vol. 15, no. 4, 2021.

[57] K. D. Albab, I. Sharma, J. Adam, B. Kilimnik, A. Jeyaraj, R. Paul, A. Agvanian, L. Spiegelberg, and M. Schwarzkopf, "K9db:Privacy-Compliant Storage For Web Applications By Construction," *USENIX OSDI*, 2023.

[58] S. Ahmad, D. Arumugam, S. Bozovic, E. Degefa, S. Duvvuri, S. Gott, N. Gupta, J. Hammer, N. Kaluskar, R. Kaushik, R. Khanduja, P. Mujumdar, G. Malhotra, P. Naik, N. Ogg, K. K. Parthasarthy, R. Ramakrishnan, V. Rodriguez, R. Sharma, J. Szymaszek, and A. Wolter, "Microsoft Purview: A System for Central Governance of Data," *VLDB*, vol. 16, no. 12, 2023.

[59] M. Ferreira, T. Brito, J. F. Santos, and N. Santos, "RuleKeeper: GDPR-aware personal data compliance for web frameworks," *IEEE S&P*, 2023.

[60] K. Alpernas, A. Panda, L. Ryzhyk, and M. Sagiv, "Cloud-Scale Runtime Verification of Serverless Applications," *ACM SoCC*, 2021.

[61] "Serverless on IBM Cloud," https://www.ibm.com/products/code-engine.

[62] "AWS CloudFormation Guard," https://docs.aws.amazon.com/cfn-guard/latest/ug/what-is-guard.html.

[63] "AWS Inspector," https://aws.amazon.com/inspector/.

[64] "Microsoft Defender," https://azure.microsoft.com/en-us/products/defender-for-cloud.

[65] A. Pimlott and O. Kiselyov, "Soutei, a logic-based trust-management system," *FLOPS*, 2006.

[66] J. DeTreville, "Binder, a logic-based security language," *IEEE S&P*, 2002.

[67] C. Fournet, A. Gordon, and S. Maffeis, "A type discipline for authorization in distributed systems," *IEEE CSF*, 2007.

[68] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modeling," *USENIX OSDI*, 2004.

[69] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," *IEEE DSN*, 2002.

[70] B.-C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities," *USENIX ATC*, 2009.

[71] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SOSP*, 2003.

[72] "AnalysisTools," https://analysis-tools.dev/.

[73] "Pyre," https://pyre-check.org/.

[74] Bearer, "Developer-first SAST for Security and Privacy," https://www.bearer.com/.

[75] I. Polinsky, P. Datta, A. Bates, and W. Enck, "GRASP: Hardening Serverless Applications through Graph Reachability Analysis of Security Policies," *ACM WWW*, 2024.

[76] W. Li, J. Ming, X. Luo, and H. Cai, "PolyCruise: A Cross-Language dynamic information flow analysis," *USENIX Security*, 2022.

[77] M. W. Alzayat, "Efficient Request Isolation in Function-as-a-Service," Ph.D. dissertation, Universität des Saarländes, 2023.

Predicate	Description
Arithmetic Predicates	
add(x, y, z)	Checks if x equals y + z
sub(x, y, z)	Checks if x equals y - z
mul(x, y, z)	Checks if x equals y * z
div(x, y, z)	Checks if x equals y / z
rem(x, y, z)	Checks if x equals y % z
Relational Predicates	
eq(x, y)	Check if x == y.
not(x)	Check if !x is true.
lt(x, y)	Checks if *x* is less than *y*
le(x, y)	Checks if *x* is less than or equals *y*
gt(x, y)	Checks if *x* is greater than *y*
ge(x, y)	Checks if *x* is greater than or equals *y*
String Comparision Predicates	
hasSubstr(str, sstr)	Checks if *sstr* is a substring in the string *str*
concat(s, x, y)	Assign to s the concatenation of strings x ‖ y.
Data based predicates	
getVal(v, d, k1, ..., kn)	Assign to v the dictionary value associated with keys k1, ..., kn, i.e., v = d[k1][...][kn].
getExternalVal(v, t, k, a)	Assign to v the value of the attribute a of the item in the table t with the key k.
Taint based predicates	
taintSetIncludes(n, l)	Check if the taint set for ADG node n contains label l.
taintSetExcludes(n, l)	Check if the taint set for ADG node n does not contain label l.

TABLE 4: Predicates for Growlithe's policy language.

Appendix A. Policy Predicates

Table 4 shows a list of Growlithe's policy predicates. These predicates are used for arithmetic, relational and string comparisons, and for retrieving values from program variables stored in Growlithe's ADG. The taint checking predicates are used for specifying flow constraints.

Appendix B. Policy Examples

B.1. Claim Processing (CP)

Here, we describe how the developer uses Growlithe to specify the policy requirements (**P1**, **P2**, **P3**) in CP. They leverage the ADG for the application shown in Figure 9. The developer updates the policy on the edge in `GetClaim`, where the *source* conduit describes an item in the `Claims` table. This policy allows read access to only the claim owner (**P1**) or the assigned adjuster (**P2**):

```
read :- (getVal(U, SessionAuth, 'sub')
    ∧ getVal(C, ObjectHandler, 'UserID')
    ∧ eq(U, C))
 ∨ (getVal(U, SessionAuth, 'sub')
    ∧ getExternalVal(A, 'ClaimAdjMap', Object, 'AdjID')
    ∧ eq(U, A))
```

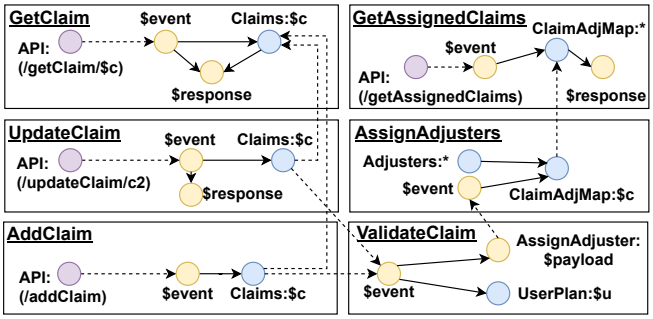

Figure 9: ADG for Claim Processing

Similarly, a developer adds the following policy in `UpdateClaim` on the edge `Claims` as the *sink* node (**P1**, **P2**). Since this is an update operation, and the policy check depends on a non-key attribute, the policy would need to fetch the required attribute for the item being updated.

```
write :- (getVal(U, SessionAuth, 'sub')
    ∧ getExternalVal(C, Resource, Object, 'UserID')
    ∧ eq(U, C))
 ∨ (getVal(U, SessionAuth, 'sub')
    ∧ getExternalVal(A, 'ClaimAdjMap', Object, 'AdjID')
    ∧ eq(U, A))
```

Finally, the developer defines the taint check policy in `AssignAdjuster` function's edge, where the *source* node is the event source in the function and the *sink* node is the `ClaimAdjMap` table. The policy from §3.2 looks like:

```
write :- taintSetExcludes(PredNode, 'UserPlan:*')
```

B.2. Image Processing (IP)

We describe how a developer specifies the policy requirements (**P4**, **P5**, **P6**, **P7**) in IP using the ADG shown in Figure 10. We only show policies specified by developers using our policy predicates and identifiers, and skip the policies which "allow" the permission.

To ensure that the functions process images in the same region (**P4**), the developer adds the following policy for each edge where the *source* node is an object in S3 bucket, and *sink* node is a local file in each function in the application:

```
read :- eq(InstRegion, ResourceRegion)
```

Next, for each edge in the intermediate functions (`Transform`, `Filter`, `Blur`) where the *sink* node is an object in S3 bucket, the developer adds the following policy to ensure that the functions store their output in the same bucket as their input (**P4**, **P7**):

```
write :- eq(InstRegion, ResourceRegion)
         ∧ concat(Label, Resource, ':*')
         ∧ taintSetIncludes(PredNode, Label)
```

And finally, the developer adds the following policy for the edge in the `TagStore` function where the *sink* node is

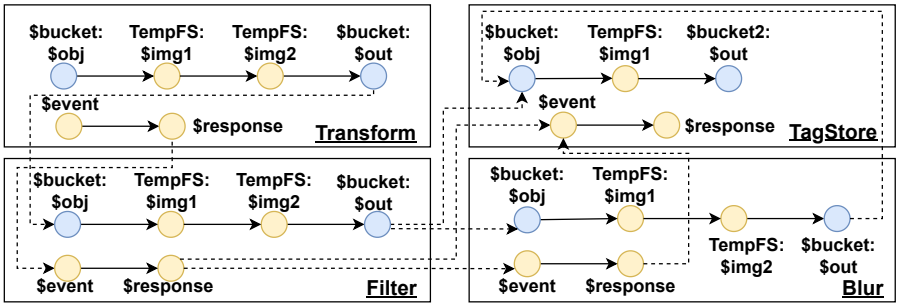

Figure 10: ADG for Image Processing

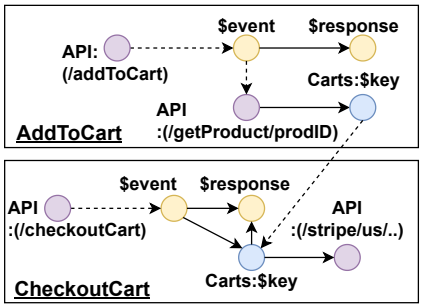

Figure 11: ADG for Shopping Cart

an object in S3 bucket to ensure that the user images are blurred and are not shared with advertisers (**P5**, **P6**):

$$\begin{aligned}\text{write :-}\ & (\text{eq}(\text{InstRegion}, \text{ResourceRegion})\\ & \wedge \text{eq}(\text{Resource}, \texttt{'FinalAdvertImgs'})\\ & \wedge \text{taintSetIncludes}(\text{PredNode}, \texttt{'AdvertImgs:*'}))\\ \vee\ & (\text{eq}(\text{InstRegion}, \text{ResourceRegion})\\ & \wedge \text{eq}(\text{Resource}, \texttt{'FinalUserImgs'})\\ & \wedge \text{taintSetIncludes}(\text{PredNode}, \texttt{'UserImgs:*'}))\\ & \wedge \text{taintSetIncludes}(\text{PredNode}, \texttt{'Blur'}))\end{aligned}$$

B.3. Shopping Cart (SC)

The developer specified policies in SC using the ADG shown in Figure 11. They specify the following policy requirement in `CheckoutCart` on the edge where the *source* node is the event source of the function, that the function should process the checkout request in the same region as the user request and the user's profile (**P9**):

$$\begin{aligned}\text{read :-}\ & \text{getVal}(\text{R}, \text{SessionAuth}, \texttt{'address'}, \texttt{'region'})\\ & \wedge \text{eq}(\text{R}, \text{SessionRegion}) \wedge \text{eq}(\text{R}, \text{InstRegion})\end{aligned}$$

Similarly, the developer specifies the policy on the edge with the *source* node as record from `Carts`, the *sink* node is an external API for processing the payment (**P8**):

$$\begin{aligned}\text{read}\ \ :-\ & \text{eq}(\text{ResourceRegion}, \text{InstRegion})\\ \text{write :-}\ & \text{concat}(\text{S}, \texttt{'https://'}, \text{SessionRegion})\\ & \wedge \text{concat}(\text{P}, \text{S}, \texttt{'.payment-gateway/process'})\\ & \wedge \text{eq}(\text{P}, \text{Object})\end{aligned}$$

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

The paper introduces Growlithe, a novel tool designed to help developers enforce data flow control policies in serverless applications. Growlithe combines static analysis with runtime enforcement to ensure compliance with data protection regulations while remaining both language- and platform-independent. It allows developers to specify and customize data flow policies declaratively and has been prototyped and evaluated on AWS Lambda applications.

C.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Provides a Valuable Step Forward in an Established Field.

C.3. Reasons for Acceptance

1) The paper creates a new tool to enable future research. Growlithe offers a novel design for specifying and enforcing data flow policies in serverless applications. By supporting cross-language and cross-service data flow policies and being made available as open-source, Growlithe promotes further research in policy compliance within serverless environments.
2) The paper provides a valuable step forward in an established field. It applies data flow control to detect compliance violations in serverless applications, an emerging setting that poses unique challenges due to the heterogeneous nature of serverless platforms. By combining static analysis and runtime enforcement, Growlithe's hybrid approach contributes to advancing data flow control in serverless environments.