SCIFFS: Enabling Secure Third-Party Security Analytics using Serverless Computing

Isaac Polinsky
North Carolina State University
Raleigh, NC, USA
ipolins@ncsu.edu

Pubali Datta
University of Illinois at Urbana-Champaign
Champaign, IL, USA
pdatta2@illinois.edu

Adam Bates
University of Illinois at Urbana-Champaign
Champaign, IL, USA
batesa@illinois.edu

William Enck
North Carolina State University
Raleigh, NC, USA
whenck@ncsu.edu

Abstract

Third-party security analytics allow companies to outsource threat monitoring tasks to teams of experts and avoid the costs of in-house security operations centers. By analyzing telemetry data from many clients these services are able to offer enhanced insights, identifying global trends and spotting threats before they reach most customers. Unfortunately, the aggregation that drives these insights simultaneously risks exposing sensitive client data if it is not properly sanitized and tracked.

In this work, we present SCIFFS, an automated information flow monitoring framework for preventing sensitive data exposure in third-party security analytics platforms. SCIFFS performs decentralized information flow control over customer data *in a serverless setting*, leveraging the innate polyinstantiated nature of serverless functions to assure precise and lightweight tracking of data flows. Evaluating SCIFFS against a proof-of-concept security analytics framework on the widely-used OpenFaaS platform, we demonstrate that our solution supports common analyst workflows (data ingestion, custom dashboards, threat hunting) while imposing just 3.87% runtime overhead on event ingestion and the overhead on aggregation queries grows linearly with the number of records in the database (e.g., 18.75% for 50,000 records and 104.27% for 500,000 records) as compared to an insecure baseline. Thus, SCIFFS not only establishes a privacy-respecting model for third-party security analytics, but also highlights the opportunities for security-sensitive applications in the serverless computing model.

CCS Concepts

• **Security and privacy → Information flow control**; **Access control**; *Distributed systems security*.

Keywords

Serverless Computing; Security Analytics; Decentralized Information Flow Control

ACM Reference Format:
Isaac Polinsky, Pubali Datta, Adam Bates, and William Enck. 2021. SCIFFS: Enabling Secure Third-Party Security Analytics using Serverless Computing. In *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies (SACMAT '21), June 16–18, 2021, Virtual Event, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3450569.3463567

1 Introduction

A key component of maintaining a good security posture is continuously monitoring your environment for security-related events. To detect threats in security event logs, it is necessary to first have threat intelligence (e.g., signatures) that can be used to generate alerts. Due to the high cost of in-house security operation centers and the complexity of security monitoring, companies are out-sourcing monitoring and threat intelligence gathering to third-party security analytics services. Not only do these service providers make security monitoring accessible to companies of all sizes, but more importantly, these providers have greater insights to global security trends than an in-house security operations center. For example, by aggregating data over all of their customers, a third-party security analytics provider can identify more threats and proactively apply responses to new threats found in one customer's environment to their entire customer base.

Unfortunately, the benefits of third-party services come with risk. Security event logs are sensitive; if data is not properly sanitized and tracked, performing aggregation over data from multiple sources can expose sensitive customer data. For example, in 2018 Facebook accidentally sent developer analytics to app testers [9]. In general, accidental data leaks are a growing concern of IT security professionals [1]. To illustrate this risk, consider a report created by an analyst who inadvertently includes sensitive data from a customer. If this report is shared with a competitor or the public, the analyst has unintentionally exposed the customer's data, potentially violating the customer's privacy or Service Level Agreement (SLA). This risk is exacerbated when data is shared with analysts who are not familiar with a company's internal procedures, an important issue as there is a push for greater collaboration between security companies in the wake of the SolarWinds hack [16]. To mitigate this risk, analysts must track data from each source client to all

SACMAT '21, June 16–18, 2021, Virtual Event, Spain

ACM ISBN 978-1-4503-8365-3/21/06…$15.00
https://doi.org/10.1145/3450569.3463567

derived reports (i.e., sinks). Manually tracking data is not only difficult and error prone, but also slows analyst workflows. Therefore, we need automated mechanisms to (1) track data and derived data and (2) prevent sharing of data without explicitly acknowledging the sources of the data and how the data was handled.

We envision a security analytics platform that allows third-parties to automatically track security-related event telemetry and derived results in order to prevent accidental data exposure. While decentralized information flow control (DIFC) [25, 32, 38, 54] is well-suited to perform this tracking and access control, incorporating DIFC into real systems often results in *label creep*, high overhead, and significant restrictions on functionality. Consider a canonical example DIFC logic where data sensitivities are represented as *tags* and data and processes are assigned *labels*, which are sets of tags. Over its lifetime, a process will read information from files and interact with other processes that have read information from other files. During this execution, the DIFC system will propagate tags from one label to another to indicate the flow of information. However, this tracking is often imprecise (e.g., conducted at the OS process layer), which causes a phenomenon commonly referred to as label creep or label explosion. Label creep is particularly problematic in server environments where long-running processes receive data from many clients with different labels. In such scenarios, the resulting over-approximated label is often meaningless and unnecessarily restricts functionality. Applied naïvely, DIFC may artificially limit the utility of third-party security analytics.

In this work, we propose SCIFFS: Securing Client Information Flows using Function-as-a-Service. SCIFFS is an automated decentralized information flow control framework for preventing accidental sensitive data exposure in third-party security analytics platforms. Our key insight for overcoming label creep is the use of serverless computing, an emerging cloud computing runtime where functionality is separated into small functions that are reentrant and ephemeral. This polyinstantiated nature of serverless computing significantly reduces the potential for label creep. However, simply tracking DIFC labels within the serverless runtime is not in-and-of-itself sufficient for a third-party security analytics platform. We augment an existing DIFC logic (Flume [32]) with workflow *caps*, which define an upper-bound for label propagation to restrict analysts to investigating only the specific customers to which they are assigned. We built a SCIFFS prototype on the popular OpenFaaS platform and evaluated it using a proof-of-concept security analytics application. Using this application, we demonstrate that SCIFFS supports common analyst workflows while imposing only 3.87% runtime overhead on event ingestion and the overhead on aggregation queries grows linearly with the number of records in the database when compared to an insecure baseline.

This work makes the following contributions:

- *We enable information tracking in security analytics workflows and prevent accidental leaks.* SCIFFS automatically tracks flows of sensitive information and allows analysts to freely aggregate data of multiple customers to identify novel threats, while preventing accidental data leaks. SCIFFS enables the auditing of any declassification event within the system.
- *We implement SCIFFS in a real-world platform.* Our SCIFFS prototype was implemented in the widely-used OpenFaaS Function-as-a-Service platform. We demonstrate how to efficiently and securely incorporate DIFC tracking within a common serverless computing runtime without dependencies on specific application languages.
- *We demonstrate the practicality of SCIFFS.* We implement a proof-of-concept security analytics serverless application and measure the performance overhead imposed by SCIFFS compared to an unsecured baseline. Our evaluation shows a 3.87% overhead on ingesting events and a linear overhead on analyzing data starting at 18.75% for a database with 50,000 records to 104.24% for 500,000 records. We further show that SCIFFS supports four common analytics workflows.

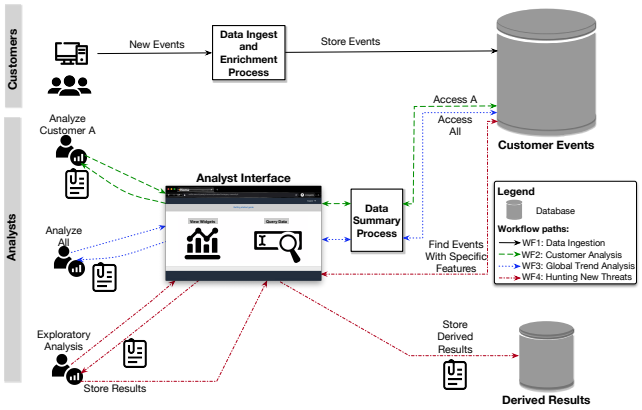

Figure 1: Common Security Analytics Workflows.

The remainder of this paper proceeds as follows. Section 2 provides background and motivation. Section 3 overviews SCIFFS. Sections 4 and 5 describe the label model and system design. Section 6 describes a proof-of-concept security analytics application. Section 7 evaluates performance overhead of SCIFFS. Section 8 overviews related work. Section 9 concludes.

2 Background and Motivation

Third-party security analytics services such as managed detection and response from Cisco [4], Cybereason [5], and FireEye [8] have become an important line of defense for enterprises. By having access to security-related telemetry (e.g., DNS requests, user login events) across many customers, these security analytics firms gain a broad perspective across entire industries. This broad perspective is critical for threat hunting, allowing the analytics firm to create new signatures that can slow down or stop threat actors as they move from target to target. A security analytics platform has three key components: (1) a software agent running on customer endpoints that reports telemetry, (2) a server-side dashboard with pre-defined widgets that summarize activity for analysts, and (3) an interface for analysts to explore telemetry data (e.g., to identify new attacks). An overview of these three components is shown in Figure 1.

Use Cases: The figure shows four common workflows that the security analytics platform must support.

WF1: *Telemetry data ingestion.* When a customer endpoint reports new telemetry, the platform may perform initial data enrichment or transformation (e.g., adding customer metadata) before storing the telemetry in a database for that customer.

WF2: *Analysis for a single customer.* One or more analysts are assigned to each customer. These analysts continuously monitor the customer telemetry for specific Indications of Compromise (IoC). This task is often achieved using pre-defined queries presented as dashboard widgets.

WF3: *Analysis over more than one customer.* Analysts are often assigned more than one customer. Dashboard widgets that aggregate statistics across customers can provide insights for threats against different industries (e.g., banking). For example, a widget may count the number of CVE exploit attempts observed in IDS logs across a set of related customers.

WF4: *Identifying new threats (threat hunting).* Skilled analysts use their knowledge and intuition to investigate scenarios that are not captured by the pre-defined queries used for dashboard widgets. In doing so, they perform *ad hoc* queries to identify new threats and build new detection capabilities. These queries commonly span multiple customers.

Problem: Without proper care, mixing data from multiple customers may either inadvertently violate SLAs or raise questions to the validity of analytics results. These concerns occur in several situations. First, sensitive customer data may be revealed to other customers when aggregated results are not properly sanitized. Second, if a customer terminates their contract, they may require all of their data be deleted, including derived results. Finally, if the analytics provider identifies a malicious customer generating fake events in an attempt to mislead aggregate analysis to report activity as benign, the analysts must identify results tainted by that customer.

Addressing these situations requires pervasive tracking of customer information. While tracking can be manually built into well-defined workflows (e.g., for dashboard widgets), unplanned ad hoc queries used in threat hunting requires automated tracking. It is unreasonable to expect the analyst to manually and accurately annotate all of the data they accessed with the correct customer information. Examples of how diverse these unplanned queries might be can be seen in open-source threat hunting "playbooks" [45].

Threat Model and Assumptions: We consider two types of adversaries: a malicious customer and a careless analyst. A malicious customer has two goals: (1) to actively generate fake events to influence analysts to view their activity as benign across all customers and (2) to passively benefit from analysts inadvertently leaking data from other customers. Given the level of data access required by analysts to perform their job, our threat model puts limits on their abilities and motivations. We primarily seek to prevent accidental data leaks. However, in the event that an analyst becomes malicious, we seek to maintain sufficient information to identify what information may be compromised or exfiltrated. All access to data is logged and we assume neither adversary can alter log files. Our trusted computing base includes the cloud platform (e.g., hardware, software, and administrators) and the code of our analytics platform solution, assuming both are free of vulnerabilities. Finally, we do not consider covert timing channels as this challenge is an orthogonal problem to cloud environments.

3 Overview

When performing security analytics across multiple customers, analysts may inadvertently expose customer information in unintended ways. This exposure may be customer data directly (e.g., specific DNS requests containing a sensitive hostname), or derived data that has not been properly sanitized (e.g., frequent DNS requests containing a sensitive hostname). The goal of this paper is to design the foundations for a security analytics platform that automatically tracks customer information in order to prevent accidental data exposure. In doing so, we must overcome the following challenges.

- *Tracking information flows in environments with long running processes (e.g., web servers) can lead to imprecision.* The longer a process is alive, the more likely it is to handle data of different sensitivities. When a process handles disjoint workflows with different data sensitivities, accurately annotating the outputs of the process becomes difficult.
- *The tracking must scale to handle data from large customer bases.* Security analytics providers are expected to have large customer bases with varying environment sizes. The infrastructure must scale to ingest data from all of the customer devices and scale to perform analysis on the collected data.
- *The platform must minimize restrictions on analysis workflows.* Improved security often limits functionality. The solution should be transparent and not impede analysis workflows.

The key insights of our solution are to (1) use decentralized information flow control (DIFC) to capture the sensitivities of different customers, and (2) implement the analytics workflow within a serverless computing runtime, also known as Function-as-a-Service (FaaS).[1] We note that serverless computing is well-suited for this purpose. First, pay-per-use and scalability are inherently part of its design, leading to the popularity of emerging commercial offerings including AWS Lambda, Google Cloud Functions, and Azure Functions. These properties make serverless computing desirable for third-party security analytics firms purely from a business perspective. Second, and more importantly, serverless computing applications are composed of small functions that are reentrant and ephemeral. From a DIFC perspective, this property provides polyinstantiation of discrete functionality within workflows. This polyinstantiation minimizes label creep. Separating functionality into smaller functions can also help tracking precision.

However, integrating DIFC into a serverless runtime is non-trivial. First, researchers have shown [44] that practical serverless deployments do not provide secure polyinstantiation. Instead, the same containers are commonly re-used for multiple function instances, a practice commonly referred to as a "warm-start" configuration. Since container re-use may permit communication outside of the DIFC model, prior approaches for enhancing serverless with DIFC (e.g., Trapeze [12]) by instrumenting the language runtime (e.g., Node.js) require "cold-start" configurations that incur significant runtime overhead to reinitialize each function instance. In contrast, our design makes the key observation that only the function handler code within the container needs to be polyinstantiated.

[1] As FaaS is the *de facto* type of serverless computing, we use these terms interchangeably throughout the paper.

Therefore, we modify the containers used by the FaaS runtime to enforce lightweight process and storage isolation *within* the container to provide secure polyinstantiation in "warm-start" configurations.

Second, propagating large labels between function instances can incur high overhead. DIFC labels typically consist of a set of tags, with each tag representing a security sensitivity. The straightforward solution of encoding DIFC labels in messages (e.g., as in Camflow [42]) works when labels are small. However, in a security analytics platform with possibly tens of thousands of customers, it is impractical to encode a set of tags in HTTP headers. The resulting process of encoding, transferring, and decoding the label for each function execution can become expensive. In the process of designing SCIFFS, we observed that our security analytics platform does not need a generic FaaS instrumentation. We found that the label of the function workflow only needs to change or be consulted at either (a) the workflow entry, and (b) storage interface. This is because functions do not need the ability to explicitly change their label in a security analytics platform and explicit label changes (e.g., raising labels and declassification) can be supported through workflow initialization. We were therefore able to optimize label propagation by encoding a workflow identifier in the HTTP header and maintaining a mapping between the workflow identifier and the current label within the storage infrastructure.

Third, implicit and explicit label propagation requires careful consideration. Early DIFC logics used explicit label propagation to avoid subtle side-channels [31]. Explicit label propagation requires the application logic to specifically define when it wants to raise its label. The raised label is then compared before an operation occurs (e.g., on read, the process label must dominate the object label). Not only are these changes to propagation logic burdensome to develop, but they also do not work well in a security analytics environment where the analyst does not know what types of data might be returned. In contrast, recent work [39] showed implicit labels (also known as "floating labels") can be safe when polyinstantiation is used. However, floating labels introduce an unintended side-effect for privileged analysts reading derived results from the database. When derived data is stored, it may be structured such that it no longer has an easy way to filter by a certain customer. Therefore, there is no way for the privileged analyst to operate on derived results for a specific subset of customers. To address this situation, we modify our DIFC logic to capture a balance between implicit and explicit label propagation. We do so with the concept of *caps*, similar to *maximum labels* in IX [35], *receive labels* in Asbestos [25], and *clearance* in HiStar [54] and LIO [49], to put an upper-bound on the workflow label. Our storage layer then automatically removes data that the does not meet the cap-policy. Thus, the analyst can never be shown data they do not want to access.

Fourth and finally, the platform must account for declassification. DIFC is a formal *approximation* of a data security policy. There are always exceptions in which an analyst may need to declassify a label (i.e., remove tags). For example, a report may no longer contain customer specific data. SCIFFS accounts for such scenarios by making declassifications *audit events*. These audit events require the analyst to justify the declassification for historical purposes.

Workflow Operation: Figure 2 depicts the high-level overview of SCIFFS and how it handles the workflows identified in Section 2. In

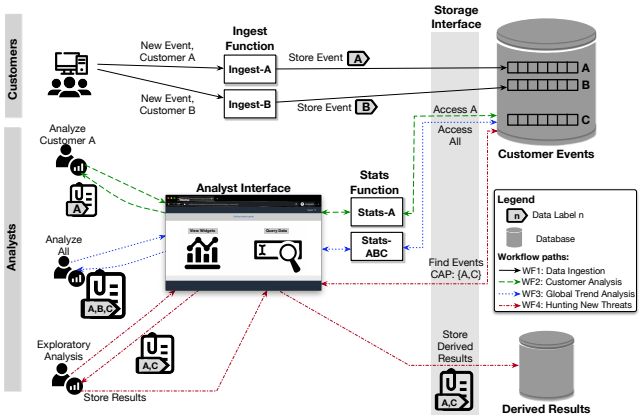

Figure 2: Security Analytics Workflows in SCIFFS.

Workflow 1, new events arrive from Customer A and Customer B. These events are received by a dedicated data ingest and enrichment function. Based on authentication credentials provided by the endpoint submitting the event, SCIFFS determines that the correct label for the received data is $\{A\}$ and $\{B\}$, respectively. Note a label is a set of tags, with each tag representing a sensitive data type (see Section 4.1). After data enrichment is complete, the ingest function stores the data to a database. In doing so, the storage interface automatically propagates the workflow label to the database by embedding it within the database row. Alternatively, the ingest function could pass the data to other functions, for additional processing, before storing to the database. Importantly, these two events are processed independently in separate function instances, preventing customer data from being mixed.

Next, Workflows 2 and 3 are initiated to perform data analytics. Common analytics routines are represented by dashboard widgets that invoke a specific pre-defined analytics function. Workflow 2 represents analysis over a single customer and Workflow 3 represents the same analysis over all customers to get a global view. Note that the workflows initially have an empty label $\{\}$. As the aggregation function executes, the workflow label accumulates tags based on the labels assigned to the data that it retrieves. For example, aggregation of Customer A's data will be labeled $\{A\}$ and the aggregation of both Customer A and Customer B's data will be labeled $\{A, B\}$. These tags are automatically added by the storage interface. Finally note, a precise and least restrictive label is returned from the database. That is to say, when aggregating over the entire database, if Customer A's data is filtered out in the query, then the label on the result will not contain Customer A's tag.

Finally, Workflow 4 shows the analyst performing a threat hunt. The analyst has direct access to the database interface and full control over the queries. As queries return results with different labels, the label of all data accessed during the hunt is tracked in a web application session. This workflow also highlights another important aspect of the SCIFFS architecture. For each query, the analyst may supply a *cap* to guarantee the results of their queries do not contain data from specific customers (unless the data was explicitly declassified previously). As such, the cap represents access control restrictions (e.g., an analyst accessing Customer B's data).

Note that SCIFFS can only track information within the cloud infrastructure and not within the client. Tracking information flow within a client is an orthogonal challenge and often requires new trusted software to be deployed on analyst systems. Therefore, SCIFFS assumes some level of trust for the analyst system, as mentioned in our threat model (Section 2). While insider attacks are a real threat, preventing a privileged analyst from abusing their access to data is out of scope for this work. However, SCIFFS enables fine-grained least privileged polices to limit the damage of malicious insiders. Fundamentally, the analyst must see some data to perform their job. Therefore, we rely on the web application to properly track the analyst's browsing session and reinstate a FaaS workflow to the next part of the web application workflow. We also assume the web application has a mechanism that allows the analyst to explicitly declassify a browsing session, providing justification. We discuss a simple example web application in Section 6.

4 SCIFFS Tracking Policy

This section defines SCIFFS's label model, including its semantics and propagation. Recall from Section 3 that SCIFFS uses a decentralized information flow control (DIFC) model, which maps nicely to the need of tracking the data for many different customers. However, we must adapt existing models for the security analytics platform environment. Specifically, we must consider (1) label propagation, (2) access control, and (3) declassification. We begin by introducing label model preliminaries.

4.1 Label Model Preliminaries

SCIFFS uses a DIFC label model inspired by Flume [32], which is based on *tags* and *labels*. A tag $t \in \mathcal{T}$ defines a secrecy class for sensitive data, where $\mathcal{T}$ is the set of all possible tags. Tags are opaque and can be assigned any semantics. Whereas Flume uses tags for both secrecy and integrity, SCIFFS only uses tags for secrecy. A label l is a set of tags in $\mathcal{T}$, where the set of all labels $\mathcal{L}$ is the power set of tags $2^{\mathcal{T}}$. For example, $l_1 = \{t_1, t_2\}$ represents a label with two secrecy tags.

Following the Flume model, each tag has two capabilities: positive (t^+) and negative (t^-). Granting a capability to a subject gives ability to add and remove a tag from a label, respectively. Any data owner can create a new tag, which gives them t^+ and t^- for that tag. The data owner may assign either capability to other processes and users, but only the data owner of a tag may distribute these capabilities. Flume uses capabilities to determine if a label change is *safe*. That a label change from l to l' is safe if the process has all positive capabilities $\{l' - l\}^+$ and all negative capabilities $\{l - l'\}^-$.

In Flume, the lattice for information flow is defined by a partial order of labels using a subset relation. Data may only flow from a process with label l_1 to a process with l_2 if l_2 *dominates* l_1 ($l_2 \sqsupseteq l_1$). Similar to Flume, SCIFFS defines dominance using subset relations: $l_2 \supseteq l_1$ implies $l_2 \sqsupseteq l_1$. The top of this lattice containing all tags is denoted as $\top$. Flume also incorporates processes' capabilities into the dominance evaluation when determining if a message flow is safe. The resulting dominance evaluation over hypothetical labels allows Flume to require explicit label changes while minimizing changes to application code. However, the hypothetical labels have the side-effect of implicitly declassifying information, which is not desirable in a security analytics setting. Instead, SCIFFS requires a floating label IFC model similar to Weir [39]. As such, SCIFFS repurposes Flume's positive and negative capabilities to define access control (Section 4.3) and declassification (Section 4.4) policies.

Finally, Flume defines the label join ($\sqcup$) operation as the union of the tags for the two labels. That is, $l = l_1 \sqcup l_2$ is equivalent to $l = l_1 \cup l_2$. SCIFFS uses the join operation both when setting the initial workflow label (Section 4.2) and when updating the workflow label when data of a new type is read (Section 4.3).

4.2 Initial FaaS Workflow Label

Recall that the serverless computing environment consists of small, reentrant, and ephemeral functions calling one another. This call chain defines a workflow. SCIFFS maintains a label l_f for each workflow within the serverless environment. When SCIFFS receives an HTTP request from outside the environment, a new workflow is created. This subsection describes how SCIFFS sets the initial serverless workflow label l_f.

The workflow label l_f is initially influenced by three sources: (1) the default label l_u for the user, (2) the web application session label l_s connecting multiple serverless workflows, and (3) a join label l_j providing additional tags specified in the HTTP request.

$$l_f = (l_u \sqcup l_s \sqcup l_j) \tag{1}$$

Each constituent label has a specific purpose within the overall functionality of SCIFFS, although depending on the particular situation some of these labels may be an empty set ({}).

The default user label l_u is primarily used when customer endpoints are reporting telemetry. That is, the user account for each customer is assigned a tag or set of tags that should be associated with the telemetry. This approach allows the platform to flexibly support a one-tag-per-customer model, or more fine-grained specification, as needed. In contrast, the default initial label $l_u = \{\}$ for analysts, as they do not commonly introduce data into the workflow. From a label model standpoint, we assume there exists a way to convert the `UserID` of the authenticated user to l_u.

Next, the serverless workflow may be part of a larger application workflow involving multiple HTTP requests from the analyst's web browser. As further detailed in Section 5.2, SCIFFS accounts for the lack of information tracking in the analyst's web browser by maintaining an application session label l_s. Conceptually, when the serverless workflow f_1 terminates and returns data to the browser, SCIFFS saves l_{f1} as l_s so that it is available when the serverless workflow f_2 begins. From a model standpoint, we assume there exists a `SessionID` for the application session that is included in the HTTP request, and there is a way to convert the `SessionID` to l_s.

Finally, SCIFFS allows the HTTP request to include a `JoinLabel` header that defines l_j. The per-request join label l_j exists to provide additional flexibility. For example, a customer may wish to specify specific custom fine-grained tags for specific endpoints. As another example, an analyst may wish to capture data flows from their own workstation (e.g., information saved in a text file) to ensure they are tracked correctly by SCIFFS. From a model perspective, we assume there exists a way to parse the `JoinLabel` from the HTTP header. We also assume that SCIFFS will deny the HTTP request if $\exists t \in l_j$ such that the user does not have t^+. We discuss the management of per-user capabilities next.

4.3 Storage Access and Label Propagation

Rather than maintaining a separate policy to define which analyst may access which customer data, SCIFFS incorporates storage access control into the DIFC model. Recall that a positive capability t^+ describes the ability to have a label containing tag t. Thus, SCIFFS uses positive capabilities to define storage access control.

Each serverless workflow f has an *effective cap* C_e that defines an upper bound (or "cap") on the tags that may float into the workflow label l_f. The effective cap C_e is determined based on a *user cap* C_u and a per-HTTP *request cap* C_r. The user cap C_u is a set of positive capabilities t^+ statically assigned by the system. Semantically, C_u defines all of the customer data types that user u should be able to access. Similar to the per-HTTP request join label used to determine the initial workflow label, SCIFFS allows an analyst to define a per-HTTP request cap C_r, e.g., to perform a query on only a subset of the customers that the analyst has access to. However, if C_r is defined, SCIFFS requires that $C_r \subseteq C_u$, denying the HTTP request otherwise. Therefore, if C_r is defined, $C_e = C_r$, otherwise $C_e = C_u$.

The SCIFFS label model considers access to each database row individually. Each row r has a label l_r. When a function attempts to read row r, SCIFFS consults the workflow label l_f. If $l_f \sqsupseteq l_r$ the read is allowed. Otherwise, SCIFFS determines if updating l_f with l_r is *safe*. To do so, SCIFFS calculates the proposed new label $l'_f = l_f \sqcup l_r$ and determines if $\{l'_f - l_f\}^+ \subseteq C_e$. If so, SCIFFS allows the read and updates l_f to l'_f. Otherwise, the read is denied and l_f is not updated. Note that when access to a row is denied, SCIFFS suppresses that row from the database results rather than rejecting the query. Therefore, the denied read does not leak information and the data in the workflow for user u will never exceed C_u.

In contrast to reads, database writes are trivial. SCIFFS assumes that the security analytics process never requires updating database rows. This assumption is reasonable because customer event data should be read-only and derived results can easily be versioned to distinguish newer results with an updated label. Therefore, SCIFFS only needs to handle `INSERT` operations. When performing an `INSERT`, SCIFFS assigns each row the current workflow label l_f.

Finally, it is useful to consider the security of a DIFC model that uses floating labels. Krohn and Tromer [31] showed that floating labels have the potential to leak information in certain situations. However, Nadkarni et al. [39] later demonstrated that when polyinstantiation is used, floating labels do not leak information. Therefore, SCIFFS's use of floating labels is secure due to the polyinstantiation provided by the serverless environment and the extensions to the floating label model (e.g., caps) further restrict a user's access to data rather than increasing their access to data.

4.4 Declassification

The final consideration for an IFC system is declassification. Traditionally, declassification to public ($\bot$ in the lattice) is needed whenever data leaves the system. However, since security analysts need to view the customer data to perform their tasks, we consider the analysts' workstations to be part of the system. Since SCIFFS does not monitor analyst workstations, it must make some assumptions about the information flows therein.

However, some analyst tasks do require declassification. For example, analysts may generate reports for customers or even public release. SCIFFS ensures that these reports are labeled based on all of the data and derived data that was used to generate them. However, in the process of generating the report, the specific contents may be sufficiently sanitized such that specific customer data can no longer be inferred (e.g., no specific IP addresses or hostnames). In such cases, explicit declassification of data is appropriate.

SCIFFS supports manual declassification if the user has the appropriate negative capabilities t^- for all t in the data label. However, SCIFFS makes each declassification an *audit event*, requiring the user to enter an explicit reason for the declassification and a description of why the declassification was appropriate. We envision several ways in which this declassification may occur in practice. For example, the declassification may be performed by a single user or a combination of users, where declassification is a serial process of individual users declassifying a subset of the tags based on their expertise. We also envision future systems could include serverless functions that are certified to remove specific tags based on customer-specific sanitization specifications, e.g., similar to data leak prevention (DLP) policies. However, for the purpose of this paper, we do not consider such privileged serverless functions.

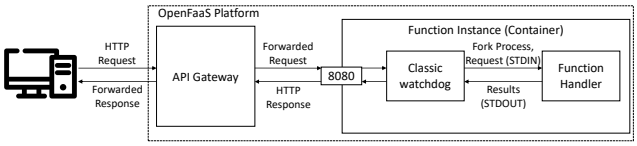

Figure 3: Overview of OpenFaaS architecture.

5 SCIFFS System Design

As described in Section 3, SCIFFS uses a serverless computing environment to overcome the label creep and tracking precision challenges historically associated with DIFC. While SCIFFS is not the first system to propose using FaaS or more general cloud computing models for DIFC tracking, prior work (e.g., Trapeze [12] and Camflow [42]) are ill-suited for the goals of the security analytics platform use case. Specifically, (1) Trapeze uses "cold-starts", which do not scale as well as "warm-start" configurations, and (2) Camflow passes labels to distributed components within the message, which can cause significant overhead for large labels. SCIFFS overcomes the first limitation by modifying the FaaS runtime to enforce lightweight isolation within containers and enable DIFC compatible "warm-start" configurations. It overcomes the second limitation by observing that a central storage of workflow labels can be efficient for the security analytics platform use case due to the situations in which the label needs to be modified or consulted.

This section describes the key system design aspects of SCIFFS: (1) label propagation between other functions and between persistent storage, (2) secure polyinstantiation in "warm-start" configurations, and (3) access control and auditing. We begin with a short background on OpenFaaS [11], which we extend to support SCIFFS. While the design concepts are transferable to other FaaS platforms, this section uses aspects of OpenFaaS to provide concrete examples.

5.1 OpenFaaS Background

Figure 3 depicts a simplified version of the unmodified OpenFaaS environment. The main components of the OpenFaaS environment are: (1) the API gateway, (2) function containers (e.g., Docker), (3) the per-container watchdog, and (4) the function handler within each container. The figure also shows clients external to the platform that interact by sending HTTP requests to the API gateway.

During normal operation, the API gateway provides an interface for creating, deleting, modifying, monitoring, and scaling functions. It is also responsible for accepting both external and internal requests and routing them to the appropriate function for processing. Function instances run within containers (e.g., Docker) with a special process called a watchdog. The watchdog listens on port 8080 within a container and accepts HTTP requests. For each request, the watchdog forks a handler process and passes the request as `STDIN` to the process. Once processing is complete, the handler process returns the results to the watchdog through `STDOUT`. The watchdog wraps the output from the handler in an HTTP response, adding any required headers, and forwards the response to the API gateway. Finally, the API gateway forwards the response to the caller. Functions can be called by external clients or other functions internal to the system. In either case the API gateway handles all requests, returning the response from the function to the caller.

5.2 Label Propagation

SCIFFS implements the DIFC label model described in Section 4. However, realizing this model into a system requires addressing two key design decisions: (1) where and how to encode the FaaS workflow label l_f, and (2) how to incorporate flows outside of the system (e.g., within the analyst workstation).

As mentioned previously, encoding l_f within the HTTP headers sent between functions may result in significant overheads when labels contain a large number of tags. Therefore, SCIFFS stores l_f in a centralized database that can be looked up using a FaaS workflow identifier. While using centralized label storage could result in significant overhead in a generic DIFC setting (e.g., checking or modifying the label on each function transition), SCIFFS only needs to consult or modify l_f during FaaS workflow initialization and when storage is accessed. Since initialization only occurs once per workflow, SCIFFS places the label storage within the storage interface, thereby optimizing label operations for storage access.

Accounting for flows outside of the system is a inherent challenge for IFC systems. As described in Section 4, the security analyst must view customer data to perform their job. Rather than continuously declassifying information to $\perp$ when sending information to the analyst workstation, we opt for a lightweight way of incorporating the analyst workstation into the system. Specifically, SCIFFS assumes that the security analytics web application maintains session semantics that can connect multiple FaaS workflows. This gives flexibility to the web application to define how HTTP requests are related to one another, as well as provide an intuitive user interface to inform the analyst of data sensitivity, and prompting for explicit declassification where appropriate. While SCIFFS could have extended DIFC into the workstation using a custom IFC enabled web browser (e.g., FlowwolF [27] and FlowFox [21]), the application session abstraction provides a reasonable balance between security and compatibility with existing software.

The remainder of this subsection details our modifications to OpenFaaS to provide (1) label initialization, (2) label propagation between functions, and (3) label propagation to and from storage.

5.2.1 Label Initialization Label initialization occurs at the API gateways that proxy all requests and responses. On each external request, the gateway first ensures the user is authenticated by checking the request's authentication token. Next, the gateway computes the effective cap C_e as described in Section 4.3, using the request supplied cap C_r if appropriate, and returning an HTTP 400 error if C_r is invalid. To initialize the label, the gateway determines l_u, l_j, and l_s. It ensures the request specified l_j is allowed based on C_e, returning an HTTP 400 error if not. If the HTTP request includes a session identifier, l_s is retrieved from the session management database. If the session identifier does not exist in the database, an HTTP 400 error is returned. If no session identifier is specified, a new session identifier is created and l_s is initialized to $\{\}$. The gateway then computes l_f using Equation 1. Before calling the first function, the API gateway adds l_f and C_e to the central label storage database. Finally, before returning the final response to the external client, the API gateway updates l_s with the current l_f.

5.2.2 Propagation Between Functions As mentioned above, SCIFFS maintains the FaaS workflow l_f in a central label database rather than encoding it in HTTP headers sent between functions. Therefore, it only needs to propagate a FaaS workflow identifier between functions. When a new request comes to the watchdog in a function container, the watchdog maps the process id (`pid`) of the new function handler to the FaaS workflow identifier in the request's HTTP header. If the handler process makes a new function request, the request is intercepted by the watchdog. Before forwarding the request, the watchdog traverses the process tree to determine the FaaS workflow identifier for the handler `pid`. When the function handler finishes, the watchdog wraps the handler output in an HTTP response as normal and removes the `pid` to workflow identifier mapping. The network mediation is enforced with a process sandbox described in Section 5.3.

5.2.3 Propagation to and from Storage The storage interface is the primary location of DIFC label propagation and access control mediation. SCIFFS uses a GraphQL [7] API over HTTP to bridge the HTTP-based FaaS environment with a backend PostgreSQL database. This API is provided by GraphJin [6], which automatically translates GraphQL statements into PostgreSQL statements. We modified GraphJin to mediate storage access and provide the desired functionality of our database interface.

When writing data to storage, SCIFFS only needs to account for `INSERT` queries, as discussed in Section 4.3. Therefore, the workflow label l_f can be stored directly with the inserted data. SCIFFS stores l_f for each database *row*. To do this, each table includes a PostgreSQL `INT[]` column for the label. The storage interface automatically modifies the original query to include l_f in the `INSERT` statement sent to PostgreSQL, where l_f is encoded as an `INT[]` representing the tags in the label. Note that the interfaces does not need to verify the request or cap, because it has already been verified by the gateway during workflow initialization.

When reading data from storage, SCIFFS modifies the `SELECT` query with an additional filter. Since the effective cap C_e defines the maximum label allowed for the FaaS workflow, SCIFFS uses PostgreSQL's `CONTAINEDBY` array operator to ensure that the query response only includes rows where the label consists of tags that are a subset of the capabilities in C_e (e.g., `WHERE` l_r `<@` C_e). SCIFFS then traverses each returned row and performs a join between the workflow label l_f and the row label l_r ($l_f = l_f \sqcup l_r$). While performing this traversal, SCIFFS also ensures to remove the label from each row in the results so it is not displayed in the result set. Once the new label is computed, l_f is updated in the central label database. One challenge with this approach is identifying a precise label for aggregate functions (e.g., `COUNT`) that do not return individual rows. To handle `COUNT`, SCIFFS performs a second query to fetch the label of all rows involved in the computation of the aggregate function and updates l_f accordingly. While IFDB [48] provides similar semantics for reads and writes, our storage interface adds support for caps, calculating precise labels on aggregate queries, and abstracts labels from the process accessing data. It is feasible to use IFDB as a replacement for the PostgreSQL database, but the additional logic implemented by the GraphJin API is still required for our design.

5.3 Secure Polyinstantiation

In traditional OS-based DIFC systems, starting two processes from the same executable provides secure polyinstantiation. In contrast, FaaS systems place the executable code for each function in a container image (e.g., Docker). In "cold-start" configurations, each time a FaaS function is needed, a new container instance is created from its image. The container instance is then destroyed when the function instance ends. Hence, cold-start configurations provide secure polyinstantiation. However, the cold-start configuration has arguably unnecessary overhead, particularly when FaaS functions are small. Therefore, many FaaS deployments use "warm-start" configurations, which do not always destroy the container instances when the function instance ends, and instead reuses the container instance for multiple function instances. As a result, malicious, compromised, or improperly defined FaaS functions can communicate outside of the DIFC model. Hence, warm-start configurations result in insecure polyinstantiation.

While SCIFFS does not explicitly handle malicious or compromised FaaS functions, there is a threat that FaaS functions are written improperly. For example, if a FaaS function writes to the container file system, that information may be read by another function instance, resulting in an accidental violation of the DIFC model. To avoid reliance on error-free FaaS functions, SCIFFS provides secure polyinstantiation for warm-start configurations.

Our key observation for providing secure warm-start polyinstantiation is that *only the function handler code needs to be polyinstantiated*. Therefore, by using lightweight isolation techniques, SCIFFS can achieve secure polyinstantiation in the same way as traditional OS-based DIFC systems. Specifically, SCIFFS modifies the OpenFaaS classic watchdog to start each function handler process in a one-time use sandbox via bubblewrap [2]. This sandbox prevents handlers from communicating directly through shared memory, indirectly through files, and all network communication is blocked except function calls through the gateway. Finally, note this work does not address covert and timing channels between function handlers, as this is an orthogonal problem.

5.4 Declassification and Auditing

The final challenge is supporting data declassification and maintaining an audit trail of data access. Declassification is the process of removing tags from the label on data and is necessary for being able to share data. For example, if an analyst is generating a report for company executives or a public blog post, that report will be labeled with the tags of each customer whose data was involved. Even if the analyst was cautious and properly sanitized the data before creating the report, the system maintains the proper label. In this scenario, the analyst can declassify the report and justify why the declassification is appropriate (e.g., removed all identifying or sensitive information). After declassification, the report can then be shared with the target audience freely.

To support declassification, we modified the API gateway and added support to the storage interface. We limit declassification to only `INSERT` operations. This is because an analyst may retrieve customer data and sanitize it locally in their browser. After storing the sanitized result, the analyst may still have access to the original data. Therefore, to ensure all future uses of the original data in a session is tracked properly we do not declassify at the browser. To declassify data, the analyst sends an `INSERT` to the storage interface and lists (1) the tags to be removed from the label and (2) a justification for why they are performing declassification. The gateway ensures the user has the appropriate negative capabilities, and if so, the request is forwarded to the storage interface. Upon receiving this request, the interface removes the supplied tags from l_f and performs the `INSERT` using the declassified label.

Finally, SCIFFS provides an audit trail of all analyst activity to aid forensics in the unlikely event that an analyst becomes malicious or their account is compromised. To support auditing, we modified the API gateway to log session identifiers, users, labels of responses, and function workflow being invoked. When declassification is preformed, the tags removed from the label and the justification is also logged. Using this information, investigators can recreate which data was impacted by a data breach. SCIFFS also supports more verbose logging such as SQL statements and the full results.

6 Proof-of-Concept Analytics Application

In this section, we describe the design and implementation of a proof-of-concept serverless security analytics application that exercises the four workflows discussed in Section 2.

6.1 Application Domain

Our analytics application collects DNS records from clients and enables analysts to create widget functions to perform common tasks, such as summarizing data, and searching for novel threats. For example, our proof-of-concept application includes a widget that summarizes DNS responses, which can be useful for identifying suspicious activity in a network. Specifically, the `NXDOMAIN` response can be used to identify malware using Domain Generation Algorithms (DGAs) to communicate with a command-and-control server. If an analyst observes anomalies in `NXDOMAIN` responses from the summary widget, it may trigger the analyst to perform a threat

hunt to see if there is a malicious cause of the anomaly. As a result of the threat hunt, the analyst may be able to define new widgets that define alerts for the newly discovered threat.

6.2 Application Implementation

The application consists of five components: a host agent, a database for persistent storage, a dashboard for viewing data, and two functions. The two functions consist of (1) an ingest function, and (2) a data summary widget function. The ingest function receives DNS logs (or batches of DNS logs) from clients, it enriches this data with customer metadata (e.g., customer account), and then stores the logs in a PostgreSQL database using the SCIFFS database interface function. The data summary function aggregates log data returning a count of how many times each DNS response code was seen over a given time interval. The data summary function can be performed for a specific customer or performed over more than one customer to identify global trends.

The two analytics functions are written in Python for the OpenFaaS platform. By design, SCIFFS is transparent to the analytics functions and the functions execute in both the unmodified OpenFaaS platform and SCIFFS without modification. We simulate customer host agents uploading batches of DNS queries to the platform using a Python script that injects DNS query logs from Mike Sconzo's DNS dataset [37]. Finally, our application simulates the analyst dashboard user interface with curl commands to invoke the analyst functions and display raw results.

6.3 Security Analytics Workflows

We now discuss how each of the four analytics workflows, described in Section 2, are covered by the application and then functionally evaluate SCIFFS in its ability to track information over these workflows in the proof-of-concept application.

The first workflow, *telemetry data ingestion* (**WF1**), is addressed by the data ingest function. To evaluate the functionality, we injected logs from different customers to the ingest function to confirm that SCIFFS correctly labeled incoming data using customer credentials and then stored the enriched data and its label in the database. Next, the second and third workflows are addressed by the DNS summary widget function. This function can be invoked for a *single customer* (**WF2**) or *more than one customer* (**WF3**). To evaluate the functionality for these workflows, we invoked the summary function multiple times for different users and different groups of users. In all cases, we observed the label on the returned data only contained the tags of the customer data involved in the computation. Finally, we address the final workflow, *identifying new threats* (**WF4**), by sending queries directly to the database interface. This query simulates an analyst further investigating the `NXDOMAIN` responses viewed in the summary function from **WF3**. Specifically, this query calculated the count of domains that resulted in `NXDOMAIN` responses over all customers. No special consideration is given to this query by the system and the query is treated as a never-seen-before task. We then store the results of this query in the database of derived data. To evaluate the functionality, we observed that the original query was labeled with the correct customer tags and that SCIFFS reinstated the correct label on the data, based on the session, when the data was stored in the derived data database.

Table 1: Average time to ingest new event. Compared to vanilla OpenFaaS, SCIFFS imposes just 3.87% overhead.

Environment	Average Time (ms)
OpenFaaS Baseline	284 ± 0.6
SCIFFS	295 ± 0.5

Note **WF4** is open-ended and is subjective to the analyst performing the threat hunt. The results of this query may contain enough information to create a new widget. For example, the data may show a pattern of domain names that indict compromised hosts using a DGA to contact the command-and-control server. However, it is possible more complex queries may be needed to identify novel threats. While GraphJin is not a replacement for SQL and is currently not as expressive as SQL, it is under constant development and future versions will add more SQL features. Alternative approaches can incorporate existing IFC databases (e.g., IFDB [48]) while modifying them to support the SCIFFS design.

7 Evaluation

In this section, we describe the experimental setup followed by an evaluation of the performance overhead imposed by SCIFFS on the proof-of-concept analytics application described in Section 6.

7.1 Experimental Setup

The modifications made by SCIFFS to the gateway, watchdog, and GraphQL API increase processing time at each component, resulting in increased request latency. Additionally, the label filters added to queries require more processing by the database server to compute the result set. To evaluate the performance overhead, we identify two key variables that impact performance of our modifications: the number of rows in the database and the number of tags in a row's label. Using these variables, we design three experiments: (1) the overhead on request latency for event ingestion (e.g., `INSERT`s), (2) the overhead on request latency when computing aggregates over the entire database with varying database sizes (e.g., `SELECT`s), and (3) the overhead imposed on a `SELECT` statement when the number of tags in a row's label is increased.

For each experiment, we make use of a 32 vCPU and 120 GB RAM Ubuntu 18.04 virtual machine for both the SCIFFS deployment and the unsecured baseline. The two VMs were hosted on separate dedicated Dell PowerEdge FC630 blade servers, each with with 2 Xeon E5-2630 2.40 GHz CPUs and 128 GB RAM running VMware ESXi 6.7.0. The unsecured baseline consists of gateway version 0.18.10, classic-watchdog version 0.18.10, faas-netes version 0.9.15, of-watchdog version 0.8.1, and GraphJin version 0.15.78. The SCIFFS environment consists of modified versions of those same code bases. Note the benchmark application is the same in each environment.

7.2 Experimental Results

We now describe each of the three experiments identified above in more detail and discuss their results.

7.2.1 Event Ingestion Overhead In the first experiment, our goal is to observe the overhead imposed by SCIFFS on processing new

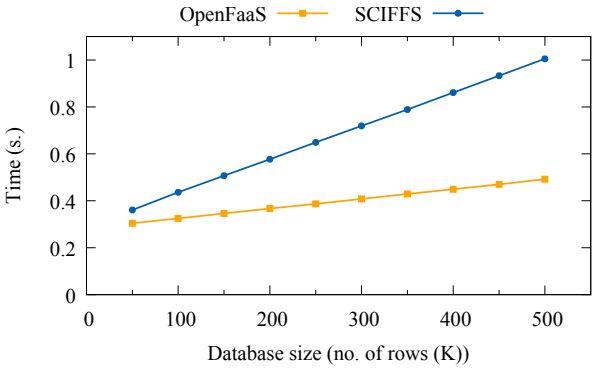

Figure 4: Average time to compute an aggregate over varying database sizes.

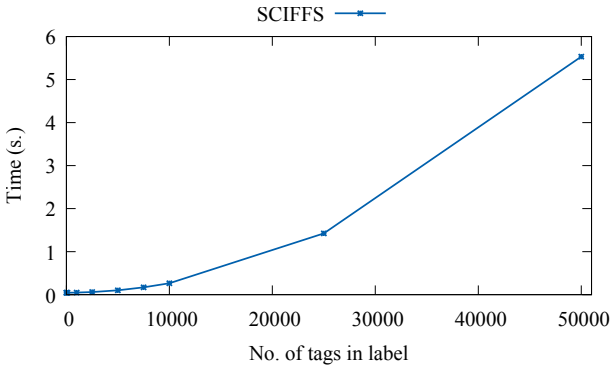

Figure 5: Average time to filter a row with varying label sizes.

events from customer devices. We note that the presence of other rows in the database do not impact the performance of the `INSERTS`, so we do not vary the size of the database in this experiment. Similarly, since data is coming from a single customer with a single tag, we do not vary the label size in this experiment. For each environment, we configured a single client to send 1,000 events to the ingest function and recorded how long each request took.

Results: Table 1 shows the SCIFFS environment imposes an 3.87% (11 ms) increase in the time to process new events. Not only is this a very negligible overhead, but the overhead on event ingestion does not impact the clients sending telemetry data because their workflow does not depend on responses from the ingest function. This experiment highlights the minimal impact on performance that results from the changes to the OpenFaaS gateway and watchdog.

7.2.2 Data Aggregation Overhead In the second experiment, our goal is to observe the overhead imposed by SCIFFS on performing aggregation over all customer data in the database. In this experiment the number of rows in the database will directly impact performance, as the label in each row of the database must be examined to compute the result set. However, we do not consider varying the size of labels for the rows, because once again we are aggregating over customer data that has a fixed single tag in the label. However, there were 3 unique labels in the data to add diversity. For each environment, we performed 10 trials configuring the database to contain from 50,000 in the first trial to 500,000 rows in last trial. For each trial, we invoked the summary function 1,000 times and recorded how long the aggregation took.

Results: Figure 4 shows the average time (seconds) taken by the summary function to compute the aggregation as the number of rows in the database was increased. For each row in the database, the label check performed by the query imposed a small amount of overhead. As the number of rows in the database is increased, this small overhead compounds and results in the linear trend observed in the figure. At 50,000 rows, SCIFFS imposed a 18.75% (57 ms) increase in the summary function execution time and at 500,000 rows there was a 104.27% (513 ms) increase. In the context of analytics workflows, these overheads are still manageable. For dashboard widgets that infrequently perform summaries over the entire database the additional processing time may not be noticed by the analyst. As databases grow into the millions of rows, it is possible to optimize these functions to execute periodically in the background and save the results to the derived database and fetch them when the analyst logs in. In situations where additional filters are supplied (e.g., filter by customer), rows can be efficiently eliminated from the result by the query optimizer first matching on the user supplied filter and then performing the expensive label check. However, given the threat analytics use case we believe these overheads to be manageable because the overhead from SCIFFS will be dominated by the longer interactive analyst process of exploring and understanding the data.

7.2.3 Overhead on PostgreSQL SELECT with Increasing Label Size In the third experiment, our goal is to observe the overhead imposed by SCIFFS on performing `SELECT` on rows with labels of increasing size. For this experiment, we considered a single row in the database and varied the size of its label. This is because in our environment, data will only have large labels when it is derived data from multiple customers. If aggregation over derived data is performed at all, the number of rows that are derived data is magnitudes fewer than the number of rows of customer data. For each environment, we directly interfaced with the database and performed 11 trials increasing the number of tags in a row from 0 to 50,000. For each trial, we performed `SELECT` statements for that specific row using a filter that verifies every tag in the label and recorded how long the query took. This query was executed 1,000 times for trials with less than 10k tags in the label and 100 times for the remaining trials.

Results: Figure 5 shows the average execution time for a query on a database with a single row with a label of increasing size. For labels with 1,000 tags or fewer, the execution takes between 45 and 48 ms. The execution takes 60 ms at 2,500 tags, then 101 ms at 5,000 tags, and more than doubles to 266 ms at 10,000 tags. At this point the execution time begins to grow much quicker, 1.4 seconds at 25,000 tags and finally 5.5 seconds at 50,000 tags. Although label encoding does not scale well for extremely large labels, we note that this is a worst-case scenario in which the tag of every customer is assigned to a row and then this row is used for further computation. To mitigate these overheads, incrementally declassifying results, when appropriate, can reduce the size of labels on derived results.

Whether this worst-case scenario could be encountered in practice depends on the client base of the analytics service and their internal procedures. SCIFFS motivates future work in scalable IFC databases, including efficient ways to encode and compute over large labels.

8 Related Work

Information Flow Control dates back to early work on the lattice model by Bell and LaPadula [15] and Denning [22, 23]. These early IFC models used a centralized definition for security labels that was maintained by a system administrator. Since this model is ill-fitting for the demands of applications, Myers and Liskov introduced the decentralized label model [38], which allowed programs to define labels and gave them permission to declassify data containing their own label. This flexible model for IFC has seen adaptions into traditional operating systems (e.g., Asbestos [25], HiStar [54], and Flume [32]), mobile operating systems (e.g., Aquifer [40] and Weir [39]), and distributed systems and cloud environments (e.g., DStar [55], Fabric [33], Camflow [42], and Trapeze [12]).

While Trapeze is the first to explicitly use IFC with FaaS, Camflow implemented FaaS-like features in their PaaS platform to obtain the properties SCIFFS and Trapeze rely on. SCIFFS differs from Trapeze by enabling warm-starts without breaking applications, supports more programming languages, and supports a SQL database compared to Trapeze's key-value store. While not a FaaS platform, Camflow replicates FaaS features by polyinstantiating a process for different labels and uses a checkpoint-restore feature to prevent label creep. However, Camflow assumes labels should not have more than five tags and optimizes their implementation on that assumption. Further, Camflow is implemented as an LSM that passes tags between distributed hosts in the request. By contrast, SCIFFS is implemented in user-space, makes no assumption on label size, and efficiently communicates labels outside of the requests, making it more appropriate for our use case. Riverbed [50] is another work that enforces user-defined privacy constraints in a distributed web application; however, this work focuses on ensuring user data is handled properly rather than performing fine-grained tracking.

Secure Multi-Execution [24] is an IFC approach that executes a program multiple times, once for each security level. Similar works include TightLip [53], Shadow Executions [17], and Ariel [18]. However, each of these works assume two security classes, public and private. Weir [39] applies DIFC to secure multi-execution by using *lazy polyinstantiation* where a process is only spawned for a label when it is needed. Further, Weir showed that floating labels do not have implicit channels when polyinstantiation is used. Since FaaS is innately polyinstantiated, our floating label model is likewise secure. Similar to multi-execution is work in *faceted values* [13]. Faceted values have two values for each variable, public and private, whose contents are determined by the user accessing the variable. Austin et al. [14] extends Jeeves [52], a Scala library that enables declaring privacy policies on sensitive values, to support faceted values. This was further extended by Jacqueline [51], which implements a web framework for faceted values in database-backed applications. Once again, we note these works only support two security classes and do not meet the demands of our use case.

Similar to faceted values are multiverse databases [34, 36] that display different views of the data, depending on the user. These works are complementary to SCIFFS but fall short in providing the desired semantics to support fine-grained information tracking SCIFFS needs. The SCIFFS storage interface is comparable to IFDB [48] in that `SELECT` statements return a subset of data depending on the process's label and `INSERT` statements use the current label. SCIFFS differs in that the label column is abstracted from processes interacting with the database, supports *caps* for `SELECT` statements, and accurately calculates a precise label for aggregate functions. Further, work into IFC databases such as the Scala-based reference monitor by Guarniereri et al. [26] and LWeb [41] provide alternatives for storing labeled data in relational databases, but once again these works do not provide the necessary semantics. SCIFFS motivates more work into efficient multiverse and IFC databases.

Serverless security is rapidly growing. Valve [20] performs network level tainting on function workflows; however, the goal is to ensure control flow integrity rather than data secrecy and is complementary to SCIFFS. SecLambda [28] provides an extensible tool for performing security tasks in serverless environments and provided three proof-of-concept tasks: flow integrity, credential protection, and DoS rate limiting. WILL.IAM [47] also provides function access control by enabling developers to specify permissible transitions of workflows. Recently there has been an increase in security analytics services (e.g., Chronicle Backstory [3], Open XDR [10]) that consolidate information from multiple telemetry sources, including external sources. These services are complementary, as they can be integrated into SCIFFS. Finally, we note works that demonstrate the benefits of serverless for data analytics (e.g., PyWren [29], IBM-PyWren [46], Flint [30], Locus [43], and funcX [19]).

9 Conclusion

This work introduced SCIFFS, an automated information flow monitoring framework for preventing sensitive data exposure in third-party security analytics platforms. We apply DIFC to serverless platforms in a novel way, to assure precise and lightweight tracking of data flows. Through a performance evaluation, we demonstrated SCIFFS has negligible overheads on processing new events and a manageable overhead for analytics queries as the dataset grows larger when compared to an insecure baseline. SCIFFS not only establishes a privacy-respecting model for third-party security analytics, but also highlights the opportunities for security-sensitive applications in the serverless computing model.

Acknowledgements

This work was supported in part by the National Science Foundation (NSF) grants CNS-1750024 and CNS-1955228. Opinions, findings, conclusions, or recommendations in this work are those of the authors and do not reflect the views of the funders.

References

[1] 2021. Accidental Data Breaches Are on the Rise; Corporate Email Is a Leading Cause. https://www.darkreading.com/attacks-breaches/accidental-data-breaches-are-on-the-rise-corporate-email-is-a-leading-cause/d/d-id/1336579.
[2] 2021. Bubblewrap. https://github.com/containers/bubblewrap.
[3] 2021. Chronicle Backstory. https://go.chronicle.security/hubfs/Backstory_WP.pdf.
[4] 2021. Cisco Managed Detection and Response. https://www.cisco.com/c/m/en_us/customer-experience/operate/managed-detection-and-response.html.
[5] 2021. Cybereason Managed Detection and Response. https://www.cybereason.com/services/managed-detection-response-mdr.

[6] 2021. GraphJin. https://graphjin.com/.
[7] 2021. GraphQL. https://graphql.org/.
[8] 2021. Mandiant Managed Detection and Response. https://www.fireeye.com/mandiant/managed-detection-and-response.html.
[9] 2021. Oops! Facebook just leaked developers' confidential data. https://www.tucsonpost.com/news/257554066/oops-facebook-just-leaked-developers-confidential-data.
[10] 2021. Open XDR - the Intelligent Next Gen Security Operations Platform. https://stellarcyber.ai/products/open-xdr-security-operations-platform/.
[11] 2021. OpenFaaS. https://www.openfaas.com/.
[12] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. *In Proceedings of the ACM Programming Languages and Systems* 2, OOPSLA, Article 118 (Oct. 2018), 26 pages.
[13] Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. 2017. Multiple Facets for Dynamic Information Flow with Exceptions. *Proceedings of the ACM Transactions on Programming Languages and Systems* 39, 3, Article 10 (May 2017), 56 pages.
[14] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*. 15–26.
[15] D. E. Bell and L. J. LaPadula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation.* Technical Report ESD-TR-75-306. Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA.
[16] Brad Smith. 2021. A moment of reckoning: the need for a strong and global cybersecurity response. https://blogs.microsoft.com/on-the-issues/2020/12/17/cyberattacks-cybersecurity-solarwinds-fireeye/.
[17] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. 2008. Preventing Information Leaks through Shadow Executions. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. 322–331.
[18] Dhiman Chakraborty, Christian Hammer, and Sven Bugiel. 2019. Secure Multi-Execution in Android. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing (SAC)*. 1934–1943.
[19] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. FuncX: A Federated Function Serving Fabric for Science. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 65–76.
[20] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing Function Workflows on Serverless Computing Platforms. In *Proceedings of The Web Conference*. 939–950.
[21] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: A Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 748–759.
[22] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243.
[23] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Communications of the ACM* 20, 7 (July 1977), 504–513.
[24] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 109–124.
[25] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 17–30.
[26] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. 2019. Information-Flow Control for Database-Backed Applications. In *Proceeding of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 79–94.
[27] Boniface Hicks, Sandra Rueda, Dave King, Thomas Moyer, Joshua Schiffman, Yogesh Sreenivasan, Patrick McDaniel, and Trent Jaeger. 2010. An Architecture for Enforcing End-to-End Access Control over Web Applications. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. 163–172.
[28] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, Thomas Ristenpart, and Michael Swift. 2020. Guarding Serverless Applications with SecLambda. arXiv:2011.05322 [cs.CR]
[29] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 445–451.
[30] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*. 451–455.
[31] Maxwell Krohn and Eran Tromer. 2009. Noninterference for a Practical DIFC-Based Operating System. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 61–76.
[32] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 321–334.
[33] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 321–334.
[34] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M. Frans Kaashoek, and Sam Madden. 2019. Towards Multiverse Databases. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. 88–95.
[35] M. D. McIlroy and J. A. Reeds. 1992. Multilevel Security in the UNIX Tradition. *Software: Practice and Experience* 22, 8 (Aug. 1992), 673–694.
[36] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. 2017. Qapla: Policy Compliance for Database-Backed Systems. In *Proceedings of the USENIX Conference on Security Symposium (SEC)*. 1463–1479.
[37] Mike Sconzo. 2021. DNS. https://www.secrepo.com/Datasets%20Description/Network/dns.html.
[38] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 129–142.
[39] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC Enforcement on Android. In *Proceedings of the USENIX Conference on Security Symposium (SEC)*. 1119–1136.
[40] Adwait Nadkarni and William Enck. 2013. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1029–1042.
[41] James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information Flow Security for Multi-Tier Web Applications. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 75 (Jan. 2019), 30 pages.
[42] Thomas F. J.-M. Pasquier, Jatinder Singh, David Eyers, and Jean Bacon. 2017. Camflow: Managed Data-Sharing for Cloud Services. *Proceedings of the IEEE Transactions on Cloud Computing (TCC)* 5, 3 (2017), 472–484.
[43] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 193–206.
[44] Rich Jones. 2021. Gone in 60 Milliseconds: Intrusion and Exfiltration in Server-less Architectures. https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds.
[45] Roberto Rodriguez. 2021. The Threat Hunter Playbook. https://threathunterplaybook.com/introduction.html.
[46] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In *Proceedings of the International Middleware Conference Industry (Middleware)*. 1–8.
[47] Arnav Sankaran, Pubali Datta, and Adam Bates. 2020. Workflow Integration Alleviates Identity and Access Management in Serverless Computing. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. 496–509.
[48] David Schultz and Barbara Liskov. 2013. IFDB: Decentralized Information Flow Control for Databases. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 43–56.
[49] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the ACM Symposium on Haskell (Haskell)*. 95–106.
[50] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-Defined Privacy Constraints in Distributed Web Services. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 615–629.
[51] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-Backed Applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 631–647.
[52] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 85–96.
[53] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. 2007. Tightlip: Keeping Applications from Spilling the Beans. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 159–172.
[54] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 263–278.
[55] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. 2008. Securing Distributed Systems with Information Flow Control. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 293–308.