IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 1 May 2025, accepted 20 May 2025, date of publication 22 May 2025, date of current version 30 May 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3572729

RESEARCH ARTICLE

BAMBDA: A Real-Time Verification Framework for Serverless Computing

CHANGHEE SHIN, BOM KIM, AND SEUNGSOO LEE
Incheon National University, Incheon 22012, Republic of Korea

Corresponding author: Seungsoo Lee (seungsoo@inu.ac.kr)

This work was supported by the Institute of Information Communications Technology Planning Evaluation (IITP) Grant funded by Korea Government (MSIT) (The Development of Darkweb Hidden Service Identification and Real IP Trace Technology) under Grant RS-2022-II220740.

ABSTRACT Serverless environments are rapidly emerging as the new paradigm for cloud computing due to their automatic scalability, cost efficiency, and ease of operation. However, IAM-based privilege management and event-driven execution mechanisms can introduce security vulnerabilities. In particular, complex inter-functional call relationships expose systems to attacks such as privilege abuse and event call condition exploitation. These attacks often occur dynamically at runtime, making them difficult to address with static defenses. Existing static analysis methods attempt to mitigate these risks, but are inherently limited in capturing dynamic attacks that occur at runtime. In this paper, we propose BAMBDA, a dynamic security framework for serverless environments that prevents privilege abuse and chained function call attacks. BAMBDA performs real-time function call verification through centralized logging and automated code injection based on application-specific log groups. Specifically, we introduce a multi-step verification process that distinguishes between direct calls, event-driven calls, and API calls, effectively preventing unauthorized attacks without requiring additional security configurations from developers. Experiments conducted in AWS Lambda environments demonstrate that BAMBDA effectively defends against privilege abuse and chained function call attacks, achieving practical deployment with minimal performance overhead of 8.12% under warm start conditions.

INDEX TERMS Serverless, cloud computing, access control.

I. INTRODUCTION

As cloud services continue to evolve, serverless computing has emerged as an event-driven paradigm that bridges the gap between the infrastructure management overhead of Platform as a Service (PaaS) and the limited flexibility of Software as a Service (SaaS). Also known as Function as a Service (FaaS), this model allows developers to focus on implementing business logic by structuring applications into independent functional units. According to a recent Datadog survey [1], 70% of AWS (Amazon Web Services) users and 60% of Google Cloud users have adopted at least one serverless solution. This widespread adoption is primarily driven by the automated execution and resource allocation managed by cloud service providers, which significantly reduces operational complexity. In addition, serverless computing dynamically scales resources in response to request traffic, ensuring that users only pay for the resources they consume, a key benefit highlighted in previous research [2], [3], [4], [5], [6], [7], [8], [9].

While the automated resource management of serverless computing improves operational efficiency, it relies on fine-grained access control between functions and resources. Identity and Access Management (IAM) serves as the primary security mechanism for managing these permissions. However, misconfigured IAM settings can result in security risks, such as unauthorized access through permission abuse or unexpected cost increases due to excessive resource consumption. Moreover, the complex inter-function call relationships in serverless environments

The associate editor coordinating the review of this manuscript and approving it for publication was Yassine Maleh.

can expose applications to vulnerabilities, including Denial of Service (DoS) [10] and Denial of Wallet (DoW) [11] attacks. According to the 2020 DivvyCloud report [12], improper IAM configurations have resulted in an estimated $5 trillion in financial losses for enterprises. Similarly, Google Mandiant's 2024 analysis [13] indicates that 30.3% of potential cloud security vulnerabilities stem from misconfigured IAM policies, making it one of the most prevalent security issues. This underscores the importance of enforcing the principle of least privilege in IAM configurations, as well as the need for a robust security mechanism to continuously monitor function workloads in real time, enabling the immediate detection of unauthorized resource access caused by permission abuse and malicious data injection during function invocations.

However, practical countermeasures to address these security challenges in serverless environments remain limited. While AWS offers IAM Access Analyzer [14] to verify excessive IAM permissions before deployment, it lacks mechanisms to detect and block dynamic security threats during runtime. Existing studies [15], [16], [17], [18], [19], [20], [21] have proposed methods such as analyzing function code and IAM policies to derive configurations aligned with the principle of least privilege or visualizing function call relationships as graphs to detect abnormal access patterns.

Although these approaches help proactively identify potential vulnerabilities, they have critical limitations in dynamic serverless environments. First, they cannot detect attacks that exploit legitimate permissions across multiple functions in complex combinations. Second, they fail to capture runtime-specific execution contexts like event-triggered function chains that only emerge during operation. Third, most existing research lacks automated security policy management, rendering it incompatible with the core principle of serverless computing, which aims to minimize user intervention. Therefore, dynamic, runtime-aware security verification mechanisms are essential to address the security gaps that static analysis alone cannot cover.

In this paper, we propose BAMBDA, a real-time security verification framework for serverless environments. BAMBDA continuously monitors function workflows to detect and block abnormal invocations in real time, ensuring a secure execution environment through a log-based, multi-stage verification mechanism. It minimizes developer intervention by automating code injection during function reconstruction, maintaining operational efficiency. Furthermore, our system accurately detects and prevents function executions caused by improper permission configurations while logging all security events for continuous security management. Our evaluation demonstrates that BAMBDA was successfully applied without errors during function reconstruction and redeployment, maintaining full compatibility with AWS Lambda [22]. Moreover, it effectively detected and blocked various unauthorized attack scenarios while incurring only a minimal performance overhead of 8.12% under warm start conditions.

In summary, our main contributions are as follows:

- We introduce BAMBDA, the first access control verification framework to leverage centralized logging for dynamically detecting and blocking real-time security threats in serverless computing environments.
- We propose an automated code reconfiguration process that analyzes the structure of serverless functions and inserts security verification code, allowing developers to apply the framework on the fly without additional security settings or code modifications.
- We demonstrate that BAMBDA effectively prevents permission abuse and chained function call attacks in AWS serverless environments by distinguishing function call conditions and applying type-specific verification processes.

The remainder of this paper is organized as follows: In Section II, we introduce the relevant background knowledge and outline the problem statement. Section III reviews related work and existing frameworks, highlighting their limitations. In Section IV, we describe the system design and overall workflow of BAMBDA. Next, automated function reconstruction, multi-stage verification, and real-time security monitoring are discussed in detail in Section V. Section VI presents the functional correctness and performance evaluation of BAMBDA. Finally, Section VII discusses the limitations and future directions of BAMBDA, and Section VIII concludes the paper by summarizing the research findings.

II. BACKGROUND AND PROBLEM STATEMENTS

A. SERVERLESS COMPUTING

Serverless computing is an event-driven computing model in which cloud service providers (CSPs) such as AWS Lambda, Google Cloud Functions [23], and Azure Functions [24] dynamically manage the infrastructure required for application execution. Users define and deploy business logic as stateless function units, with each function executing independently without maintaining state. Unlike traditional server-based environments, where servers must be provisioned in advance to ensure predictable costs and performance, serverless environments dynamically allocate resources on demand, executing functions only when required and automatically releasing resources when done.

Generally, the structure of serverless functions shares core components across different platforms. The *Handler* serves as the entry point of the function, containing the main logic responsible for receiving and processing event data. The *Runtime* defines the language-specific execution environment, supporting languages such as Node.js, Python, and Java. The *Event* specifies the event source that triggers the function's execution. Configuration details, such as database connection information or API keys, can be externally injected via environment variables, which manage these parameters. Lastly, the *Policy* defines the resources and permissions accessible to the function. These components are typically specified in template files using YAML or

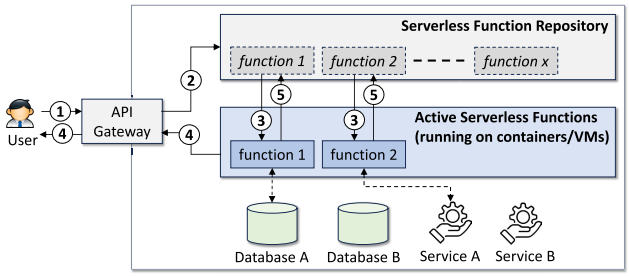


FIGURE 1. General workflow of a serverless function execution.

JSON formats, simplifying the configuration of the function's deployment and execution environment.

Figure 1 illustrates the basic workflow of serverless computing. When a user sends a request to a specific API endpoint, the request is transmitted to the API gateway (or event router) (1). The gateway invokes the appropriate function based on predefined routing rules (2). The function is instantiated as a container or virtual machine (VM), processes the input, and, if necessary, accesses external storage or a database for data operations (3). Upon completion, the result is returned to the user via the gateway, allowing the user to retrieve the response without managing server configurations or scalability concerns (4). After a predefined period of inactivity, the container is deactivated (5). The cloud provider handles infrastructure operations such as resource provisioning, container orchestration, function isolation, and scalability, enabling developers to focus solely on application logic.

Each serverless function executes within an independent container, and the platform manages the container lifecycle by distinguishing between two states: cold start and warm start. A cold start occurs when a new container is initialized, requiring runtime environment configuration and introducing latency. In contrast, a warm start reuses a previously allocated container, enabling faster execution by eliminating initialization overhead. The serverless platform handles function concurrency through auto-scaling, dynamically provisioning or reclaiming containers based on function call patterns and idle periods.

B. SERVERLESS FUNCTION INVOCATION AND IAM

Synchronous invocation, a fundamental method for calling serverless functions, involves a client sending a request and waiting for a response. As illustrated in Figure 2, when an HTTP request is received through the API Gateway, it is routed to a Lambda function, which verifies IAM authorization while executing its logic. If the request is authorized, the function processes the request using a handler within a specific runtime environment, such as Node.js. During the execution, the function may access external resources, such as a database (i.e., TestDB), if required. Upon completion, the result is immediately returned to the client via the API Gateway. Synchronous invocation is commonly used in web applications that require real-time responses, such as user authentication and data retrieval.

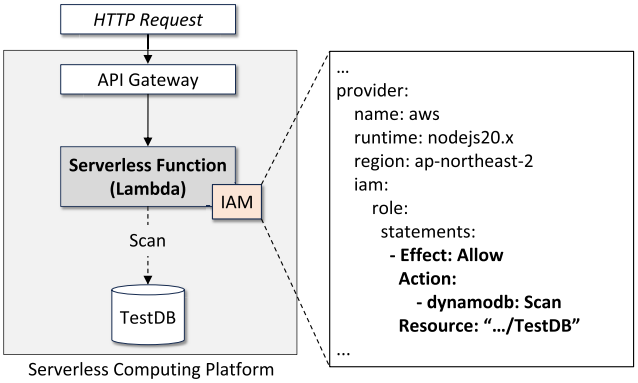


FIGURE 2. The example of IAM role permission validation in the lambda serverless function.

In contrast, asynchronous invocation immediately acknowledges a request without waiting for the function execution to complete. This method is used in scenarios where an immediate response is unnecessary, such as data processing or notification transmission. For example, when an event occurs, such as a file upload to AWS S3 or the publication of an SNS message, the service triggers an AWS Lambda function, which can subsequently forward the execution result to another service. Finally, stream-based invocation is used for continuous data processing, such as real-time analytics or log processing.

Each invocation method requires a distinct permission management mechanism, with fine-grained access control applied according to the principle of least privilege. For synchronous invocations, the API Gateway must have permission to invoke the Lambda function. In asynchronous invocations, the event source (e.g., S3, SNS) triggers the function using a role with Lambda execution permissions. For stream-based invocations, read permissions to the stream source are required, along with write permissions to store the processed data if necessary.

Identity and Access Management (IAM) is a representative authorization control mechanism in serverless environments. As illustrated in Figure 2, IAM authorization policies are defined in a structured format and consist of three main components: `Effect`, `Action`, and `Resource`. The Effect specifies whether a permission is allowed or denied, while the Action defines the operations that can be performed, such as dynamodb:Scan. The Resource indicates the name of the accessible resource. IAM policies are associated with roles, which are then assigned to functions. When a function is executed, it obtains temporary security credentials through the assigned role, enabling access to the required cloud services.

C. PROBLEM STATEMENTS

In serverless computing, IAM policies serve as the core security mechanism for controlling access between functions and resources. However, the dynamic nature of serverless environments, including function event chaining and complex

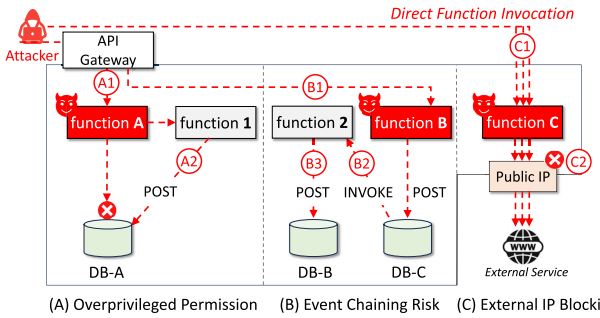


FIGURE 3. The three scenarios of serverless function abuse. Functions 1 and 2 are normal serverless functions, while functions A, B, and C are malicious ones.

function call relationships, can create vulnerabilities to unauthorized attacks. As illustrated in Figure 3, this study assumes the presence of malicious functions within an AWS Lambda application and identifies three key security challenges by analyzing common unauthorized IAM attack patterns.

C1) Overprivileged API Permissions. Serverless functions are granted permissions to interact with other AWS services through IAM policies. However, for development convenience, excessive permissions are often assigned using wildcards(*) [25]. Figure 3(A) illustrates an unauthorized access scenario resulting from such permission abuse. In this scenario, malicious Function A lacks direct access to DB-A. However, due to the granted `lambda:InvokeFunction` wildcard(*) permission, it can invoke any Lambda function. An attacker can exploit this by first calling Function A (A1), which then indirectly invokes Function 1, ultimately allowing a POST request to DB-A (A2). This unauthorized access can lead to data leakage or tampering. While some cloud security services attempt to mitigate such risks, they primarily rely on static analysis, which is insufficient for real-time threat detection in dynamic serverless environments and incurs significant costs.

C2) Risk of Event Chaining Exploits. Serverless functions can be linked through event chaining, where one function automatically triggers another when a specific event occurs. However, insufficient verification of execution conditions between the event-generating function and the event-handling function makes this mechanism vulnerable to malicious event trigger attacks. Figure 3(B) illustrates a vulnerability exploitation scenario involving database manipulation events. In this case, when Function B registers new data in DB-C, Function 2 is automatically invoked to process additional information. An attacker can exploit this chaining relationship by repeatedly sending excessive data requests through Function B (B1). Since each request triggers Function 2 without any event validation mechanism (B2), this leads to uncontrolled chain calls. Such an attack can result in excessive resource consumption, increased operational costs, and potential damage to database integrity (B3). Moreover, because this attack directly exploits legitimate event chaining mechanisms, it is difficult to detect using standard IAM policies.

C3) Unrestricted Direct Function Invocation. In addition to IAM policies, security mechanisms such as API Gateway and VPCs are used to control access to serverless functions. However, these mechanisms often fail to account for the context in which a function is directly invoked. Figure 3(C) illustrates an attack scenario [26] that exploits this limitation. Serverless functions use shared external IP addresses provided by cloud vendors when communicating with external services. To exploit this, an attacker invokes Function C (C1) directly, disguising the Origin and Referer fields in the HTTP header to mimic legitimate web page requests. Function C then sends multiple malicious requests to the external service. As a result, the external service detects a high volume of suspicious requests originating from a specific public IP and blocks the address (C2). This results in a denial-of-service (DoS) attack that prevents legitimate users sharing the same public IP from accessing the service.

Addressing C1 (Overprivileged API Permissions) using IAM requires analyzing the workflow of serverless functions within the application and configuring IAM policies for each function according to the principle of least privilege. However, this process involves extensive user intervention and significant time investment. Without automation, applying such policies at scale is impractical. Furthermore, since IAM primarily serves as an access control mechanism, it does not verify the necessity or validity of function call events. Consequently, C2 cannot be fully resolved using IAM alone. Lastly, C3 cannot be mitigated solely through IAM due to the inherent infrastructure and architectural constraints imposed by cloud vendors.

Threat Model and Assumptions. Based on the security challenges identified above, we define a threat model that outlines attacker capabilities and limitations. In this model, we consider an attacker who has compromised at least one serverless function within the application. The attacker possesses the following capabilities: (1) ability to modify the source code of the compromised function to embed malicious logic, (2) exploitation of overprivileged IAM permissions to access unauthorized resources through function chaining (exploiting C1), (3) generation of malformed events to trigger uncontrolled execution chains (exploiting C2), and (4) crafting of spoofed requests with manipulated headers to exploit IP-based vulnerabilities (exploiting C3). However, the attacker operates under specific limitations: they cannot modify IAM policies directly, alter other deployed functions, or compromise the cloud infrastructure.

The attack surface in our model encompasses interfunction invocation paths via AWS Lambda APIs, event-driven triggers through services like DynamoDB streams, API Gateway endpoints, and shared cloud resources such as databases and external services. We assume the AWS control plane and core IAM services themselves remain trustworthy, focusing instead on the misuse of legitimate permissions

TABLE 1. Comparison of related work on serverless security framework.

References	Invocation Type Analysis	Function Flow Analysis	Dynamic Verification	Multi-layered Verification	Automation
Polinsky et al. (GRASP)	×	×	×	×	×
Gupta et al. (Growlithe)	○	×	○	×	○
Padma et al. (DAuth)	×	×	○	×	×
Wen et al. (SlsDetector)	×	×	×	×	×
Jegan et al. (Kalium)	○	○	○	○	×
Kumari et al. (WAM)	○	○	×	×	×
Polinsky et al. (SCIFFS)	○	○	○	×	×
Alpernas et al. (Trapeze)	○	○	○	×	×
Datta et al. (ALASTOR)	○	○	×	×	×
Ashkenazi et al. (SMART)	○	○	×	×	×
This Work (BAMBDA)	○	○	○	○	○

and services. BAMBDA specifically targets unauthorized access patterns occurring within these components, while recognizing that physical infrastructure compromises, supply chain attacks, and direct AWS account compromises remain outside our scope. Given these assumptions about the threat landscape, effectively addressing these security challenges requires advanced dynamic analysis techniques and policy enforcement mechanisms.

III. RELATED WORK

In this section, we review recent research in serverless security, focusing on static analysis and dynamic verification approaches as summarized in Table 1.

A. STATIC ANALYSIS FOR IAM AND SERVERLESS

With the increasing security threats caused by permission misconfigurations in serverless functions, research on static analysis of IAM policies [18], [27], [28] and source code analysis of function logic [29] has gained significant attention. Grasp [18] uses static analysis of IAM policies to visualize the allowed call paths between serverless functions as a graph. By identifying all possible interaction paths, it enables the detection of potential threat vectors in advance. SlsDetector [27] utilizes large language models (LLMs) to analyze the configuration files of serverless functions, automatically generating and verifying IAM policies that adhere to the principle of least privilege. DAuth [28] enhances security by issuing access tokens to clients and managing function execution authentication through a dedicated authorization server, effectively preventing unauthorized users from accessing sensitive data.

Growlithe [29] analyzes function source code to visually represent the data flow graph. By applying permission labeling to each function based on this graph, it regulates access flow and prevents the execution of functions that violate permission constraints. SecLambda [19] performs permission analysis on function code to identify accessible functions and resources. It then constructs a graph representation of these relationships, detecting and blocking exceptional function flows and anomalous requests in real time. In addition to these research efforts, cloud vendor security services also provide IAM policy analysis. For example, AWS IAM Access Analyzer [14] performs static analysis of IAM policies to identify overly permissive configurations and issue warnings to developers.

However, as shown in Table 1, while some existing tools provide invocation type analysis or function flow analysis, static analysis-based approaches have a fundamental limitation: they cannot capture the complex function call relationships that occur at runtime in dynamic serverless environments. In serverless architectures, functions are triggered by various events and execution contexts, making it difficult to detect and mitigate real-time attacks using static analysis alone. To address this limitation, most static analysis tools require developers to manually define detailed permission settings and execution constraints for each function. However, this approach imposes a significant operational burden on developers, making it difficult to effectively manage security at scale, particularly due to the lack of automation in these systems.

Unlike previous studies, BAMBDA addresses the limitations of existing tools by allowing users to comprehensively monitor both function calls and event-level behavior in real time, without additional configuration, based on static analysis. Specifically, BAMBDA performs static analysis of function code to identify resource access patterns and interfunction call relationships. Based on this analysis, it automatically inserts the necessary logging code at appropriate places in the existing code without affecting its functionality.

During execution, it tracks the complete workflow of all functions through centralized logging, effectively detecting dynamic threats such as permission abuse or abnormal event chains that cannot be mitigated by IAM policies or traditional static analysis alone. This approach reveals critical security insights that remain hidden in traditional monitoring systems, enabling organizations to proactively

identify potential vulnerabilities and respond to threats before they can be exploited, thereby strengthening their serverless security posture.

B. DYNAMIC VERIFICATION FOR SERVERLESS

In addition to static analysis, runtime workflow verification research has been proposed to address the dynamic characteristics of serverless environments [30], [31], [32]. WAM [30] dynamically inserts tokens and roles into request headers during workflow execution to verify interfunction call relationships, enforcing role-based access control (RBAC) policies to prevent unauthorized access. ALASTOR [31] generates a provenance graph by tracking system calls and network traffic of serverless functions in real time, enabling the detection and blocking of malicious activities. SMART [32] constructs an event graph based on interaction logs between functions and resources to detect anomalous function calls in real time.

From a data flow control research perspective, SCIFFS [16] applies labeling to both functions and data to regulate the flow of sensitive information, ensuring policy-based data security even in multi-tenant environments. Trapeze [21] prevents leakage of sensitive data by employing dynamic information flow control techniques and security shims, and verifies the integrity of data transmission between functions within a sandboxed environment. Kalium [17] detects anomalies in data flow by leveraging a control flow graph generated by manual tagging and tracing.

While these methods are effective at detecting attacks based on data flow, they have a fundamental limitation: they are highly dependent on the accuracy of the labeling or tagging. As a result, they are susceptible to false positives and over-detection due to missing or incorrectly assigned tags or tokens in complex workflow environments.

In addition, as shown in Table 1, most dynamic verification approaches lack multi-layered verification and automation capabilities. Verifying only function connection relationships makes it difficult to detect unauthorized attacks or abnormal event chains disguised as normal workflows. This is because dynamic features such as event triggers and chain calls are not fully incorporated into the analysis. In particular, graph-based analysis struggles to accurately capture the execution order and call context of functions. Finally, as system complexity increases, the likelihood of missing actual attack patterns also increases due to the lack of specialized verification for different invocation types.

By contrast, BAMBDA introduces a deep verification mechanism for each function call type (i.e., API, event, and direct invocation), enabling precise analysis of execution context and event flow while automatically applying customized verification procedures. By specializing verification for each call type, BAMBDA examines detailed execution contexts, including the function's caller information, request headers, event payloads, and execution timing. This allows for a more effective distinction between normal and abnormal call patterns. Compared to existing labeling- or token-based verification methods, this approach provides a more sophisticated response to complex attack patterns while significantly reducing false positives and over-detection.

IV. BAMBDA OVERVIEWS

This section outlines the design considerations underlying BAMBDA and provides a comprehensive overview of its system architecture. In short, the system enhances the security of serverless environments by integrating real-time function call validation with automated code injection.

A. DESIGN REQUIREMENTS

This study is based on the hypothesis that IAM policy-based authorization management alone is insufficient to fully ensure dynamic workflow security in serverless computing environments. Therefore, verifying invocation conditions at the time of function execution is essential. Given the complex interfunction invocation relationships and event-driven execution of serverless environments, integrating IAM policies with dynamic function execution verification is expected to more effectively mitigate unauthorized attacks. To address the challenges discussed in Section II-C and validate this hypothesis, this study proposes the following three key design requirements.

R1: Centralized Function Workflow Tracking. A centralized monitoring system should be implemented to track and manage the workflow of all serverless functions. This system should create a unified log group for each application to comprehensively record the call information of distributed functions. Additionally, it should log function call data on a per-user basis and differentiate duplicate calls by assigning a unique identifier to each function. This mechanism enables real-time tracking of chained function executions, ensuring precise monitoring and analysis of serverless workflows.

R2: Automated Verification Code Injection. An automated code reconstruction mechanism should be implemented to analyze serverless function code and seamlessly integrate security verification code. Specifically, the system should automatically analyze each function's structure and call path to identify optimal insertion points, enabling the verification of request sources and behavior patterns without disrupting the function's original execution. This mechanism should apply a consistent level of security verification across all functions without manual intervention. Additionally, the modified function code should be automatically redeployed to the serverless environment, ensuring that security enhancements take effect immediately.

R3: Real-time Verification by Invocation Type. A sophisticated verification mechanism should be implemented to accurately detect and block hidden threats based on the type of serverless function call. For direct calls made by functions with excessive permissions, the system should verify the function's existence and the scope of its permissions. For indirect function calls, the system should analyze the request's source and contextual validity to identify abnormal call patterns or disguised requests. Finally, for chained calls

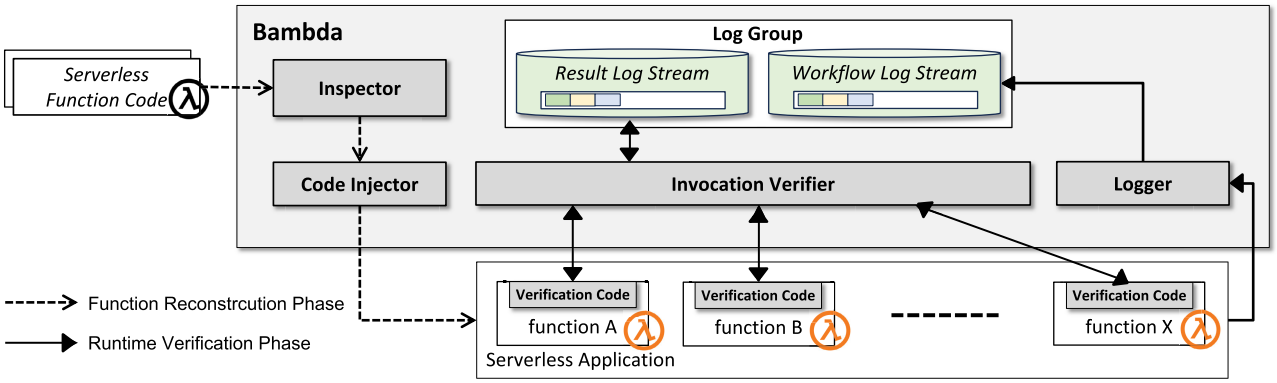


FIGURE 4. Overall architecture of BAMBDA with four key components: (i) Inspector, (ii) Code Injector, (iii) Logger, and (iv) Invocation Verifier. The workflow is divided into two phases: function reconstruction and runtime verification.

triggered by event chaining, the system should verify both the event source and data validity to effectively prevent malicious invocations.

B. SYSTEM ARCHITECTURE

As illustrated in Figure 4, the framework comprises four main components: Inspector, Code Injector, Logger, and Invocation Verifier. It operates in two distinct phases: the function restructuring phase and the runtime verification phase. This integrated approach enhances security by enabling continuous monitoring and validation of function execution.

The *Inspector* analyzes function code to identify key entities and determine security verification requirements. It first examines the input function code for resource entities, such as databases or storage services. Next, it extracts critical information, including event trigger types, function entry points, and resource access patterns from both serverless functions and their configuration files. Based on this analysis, the Inspector assesses the required level of security verification for each function and automatically configures an allow list by identifying interfunction call relationships.

The *Code Injector* automatically inserts three types of code based on the Inspector's analysis, ensuring enhanced security while preserving the function's original logic. First, it injects logging code to record execution logs whenever a function is called. Second, for functions that interact with external resources, it adds verification code to validate access permissions before allowing resource interactions. Third, it integrates identification-passing code, enabling the calling function to include its identification information in the event payload, thereby preserving the call origin. These modifications ensure comprehensive function tracking and secure resource access without altering the function's intended functionality.

The *Logger* creates an independent log group for each serverless application, enabling centralized management of all execution-time events. It integrates execution data from distributed functions using the Workflow Log Stream, providing a unified view of function interactions. Specifically, it conducts real-time analysis and verification based on all events and resource access records throughout the function's execution. The analysis results are then stored in the Result Log Stream for further evaluation and security enforcement.

The *Invocation Verifier* manages independently operating verification codes, each designed to perform specialized verification based on the call type. For calls that exploit excessive permissions, it verifies the caller's authority and intent to regulate access. For direct impersonation attempts via public IPs, it validates the request's source and context to detect unauthorized access. Additionally, for chained calls triggered by event chaining, verification is performed based on the event's data structure and call pattern. The results of all verification processes are recorded and managed within independent log streams for each verification code in the log group, ensuring comprehensive monitoring and analysis.

Workflow. BAMBDA processes user-provided serverless function code in two stages: function reconstruction and runtime verification. First, all input functions are analyzed by the Inspector, which identifies resource entities and examines call relationships to configure an allow list. After the analysis is complete, the Code Injector inserts the required security verification code into the function. The modified function is then deployed and prepared for execution.

During the runtime verification phase, the Logger records all processes from function start to event call at the entry point. These logs are stored in the log group to support the subsequent verification phase. If a function does not interact with external resources, its original logic is executed immediately after logging. However, for functions that access resources, the Invocation Verifier performs runtime verification. This process validates the execution context by analyzing the source and type of the function call and assessing resource access permissions based on the configured allow list. The function executes its original logic only if the call is deemed valid; otherwise, execution is immediately blocked if a security issue is detected.

V. BAMBDA SYSTEM DETAILS

This section explains how BAMBDA implements centralized function workflow monitoring, automated function restructuring, and call type-specific validation in AWS Lambda [22] to meet the design requirements outlined in Section §IV-A.

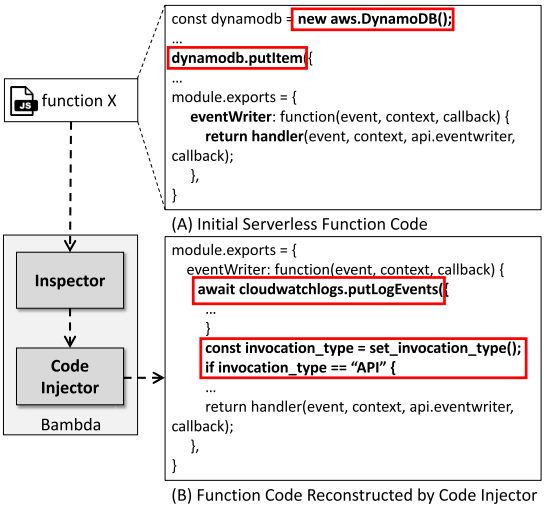


FIGURE 5. The example of automatic function restructuring performed by the Inspector and Code Injector.

A. CENTRALIZED FUNCTION WORKFLOW MONITORING

In AWS Lambda, logs are created and recorded separately for each function, resulting in distributed data that is challenging to analyze. To address this, BAMBDA leverages log collection and analysis services such as AWS CloudWatch [33] to establish a centralized logging system. Specifically, it creates an independent log group for each serverless application, enabling unified management of invocation data across all functions. For example, when a serverless application named Test-App is deployed, the system generates a dedicated log group called Test-App-Log-Group, where invocation records from all associated functions are aggregated and centrally managed.

Next, a *Workflow Log Stream* is created for each function within its respective log group to track the entire lifecycle of the function call. All events from the start to the end of each function execution are sequentially recorded, along with detailed information about resource access. For example, when a function interacts with services from the same cloud vendor, the log captures basic execution details, including call time and method type (e.g., POST, GET). If the function accesses a database, additional details such as query execution time, the number of affected records, and total operation time are recorded. For calls involving external services, communication-related data, such as HTTP header values and request/response times, are logged.

In addition to basic tracing, our system enhances function call tracking by embedding caller identification information (e.g., function name and request ID) into the event payload, which is then passed to the callee. For calls involving external services, a security token is also included. This approach enables cross-referencing of function executions, allowing the system to trace links between the caller and callee by recording request IDs within the Workflow Log Stream. For instance, when one function invokes another, the request ID from the caller's payload is logged alongside the callee's execution details, providing full visibility into the execution flow of chained calls.

B. AUTOMATED FUNCTION RECONSTRUCTION

A key limitation of AWS Lambda's default logging mechanism is that logs are recorded only after the function has fully terminated, which may result in missed critical events during execution. To address this limitation, our system automatically identifies key verification points and inserts additional security code while preserving the function's original logic. This approach enables real-time tracking and verification of the entire execution process, ensuring that critical security events are captured as they occur.

Figure 5 illustrates the function reconstruction process. First, the system identifies resource entities by analyzing service object declarations (e.g., new aws.DynamoDB()) in the handler.js file and corresponding method calls (e.g., dynamodb.putItem) within the AWS SDK. Resource entities include external service calls such as requests.post(), https.request(), and fetch(), as well as AWS services like DynamoDB, S3, SQS, and SNS. For functions that interact with external services, a security token is embedded within the API call payload in the front-end code during the reconstruction step. This token provides essential verification data, allowing the system to distinguish between legitimate and malicious requests during the request validation phase, particularly in API Gateway environments. Finally, the system analyzes function invocation paths specified in the events field of the serverless.yml file (e.g., http, stream) to configure an allow list. For Step Functions involving chained executions, the policy constructs an allow list based on the states:* entity, ensuring secure function-to-function communication.

Based on this analysis, three core code insertions are applied to each function. First, logging code is inserted into all functions to record essential execution details, including the function name, request ID, and call time at the start of execution. Second, for functions that interact with resource entities, additional verification code is injected to validate execution appropriateness each time a function call or event occurs. Third, these functions also include code to append their own identification information (e.g., request ID, function name) to the event payload, enabling call relationship tracking across functions. Because the verification logic interacts with external services such as DynamoDB and CloudWatch Logs, this may introduce latency depending on the complexity of the function's structure. To address this, we designed the verification mechanism to perform these interactions concurrently during runtime, minimizing performance impact by overlapping network-bound operations.

As depicted in Figure 5(B), all injected code is placed before the function's original logic. To achieve this, BAMBDA identifies entry points in the function code, such as module.exports and function() declarations, and locates the return

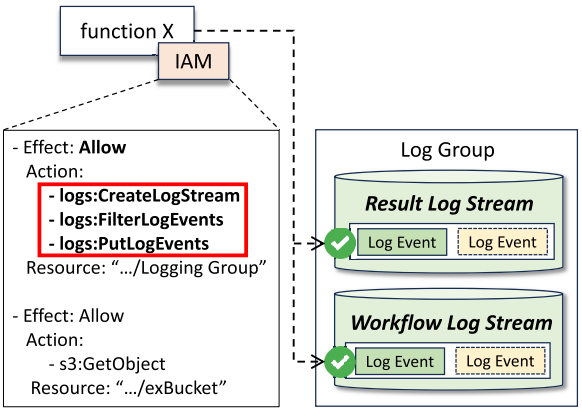


FIGURE 6. The example of the extended IAM role permission for verification.

statement to determine the complete function structure. BAMBDA rewrites the function by constructing its abstract syntax tree (AST) and injecting instrumentation blocks—including logging and verification logic—before the first executable statement. This AST-based rewriting ensures that all runtime checks are executed prior to business logic while preserving the function's semantics, scope, and control flow. Based on this structural analysis, all security-related code is inserted at the beginning of the function, ensuring that logging and verification are executed before the original business logic.

Extended IAM Role Permission. The security mechanism of BAMBDA extends IAM policies, as illustrated in Figure 6, to grant controlled access for service interactions. First, access to AWS CloudWatch is essential for real-time monitoring. Permissions such as logs:CreateLogStream (to create log streams) and logs:PutLogEvents (to record log events) are granted to enable centralized management of function call information. Additionally, access is granted to both the source service (which invokes the function) and the destination service (which is executed) to facilitate function call verification. For direct function calls, the Verifier Log Group must confirm whether the function is invoked by a valid source function, which requires the logs:FilterLogEvents permission. For event-driven function executions, access to the source service is necessary to evaluate event reliability. Similarly, for functions triggered by external service calls, the input data should be compared and analyzed against the expected data type of the destination service to ensure data integrity, while also requiring access to the destination service.

C. INVOCATION TYPE-SPECIFIC RUNTIME VERIFICATION

BAMBDA deploys automated security verification code for each function, ensuring immediate security validation based on the allow list whenever a function call or event occurs. For direct function calls, verification begins by confirming the existence of the invoked AWS Lambda function. This is achieved by comparing the function name and request ID from the received payload with real-time records in the Workflow Log Stream to verify actual execution. Additionally, real-time logs are analyzed within a one-second verification window to account for delays caused by function initialization and container provisioning. This threshold helps distinguish between normal function execution and intentional retry attempts. Once the function's existence is confirmed, the system assesses the validity of the function call by comparing the function name in the payload with the allow list. If the names do not match, the call is classified as an unauthorized attack and is immediately blocked.

Algorithm 1 Type-Aware Function Invocation Verification

Input: Invocation Event E
Output: 0 or -1

```
record_current_event()
invoke_type ← get_invocation_type(E)
if invoke_type = Direct then
    allow_list ← get_allow_list()
    invoking_func ← get_invoking_func(E)
    exist_invocation ← check_existence(invoking_func)
    if exist_invocation = FALSE then
        return -1
    if invoking_func ∉ allow_list then
        return -1
if invoke_type = Event then
    event_data ← get_event_data(E)
    is_allowed ← check_event_auth(event_data)
    if is_allowed = FALSE then
        return -1
if invoke_type = API then
    request_data ← get_request_data(E)
    security_token ← get_security_token(request_data)
    if security_token = NULL then
        return -1
    is_allowed ← check_request_auth(request_data)
    if is_allowed = FALSE then
        return -1
return 0
```

Figure 7 illustrates the verification process when `function Y` directly invokes `function X`. When `function X` is called (1), a call event is generated, triggering the verification code (2). The system first identifies the call as a direct invocation (3) and verifies the function's existence by checking the function name (`function Y`) and request ID (`ksi23bas-4cz3-mzal-flv2-1fsxzmfusds`) against real-time execution records (4). Next, it determines whether `function Y` is included in the allow list for invoking `function X` (5). Since `function Y` is

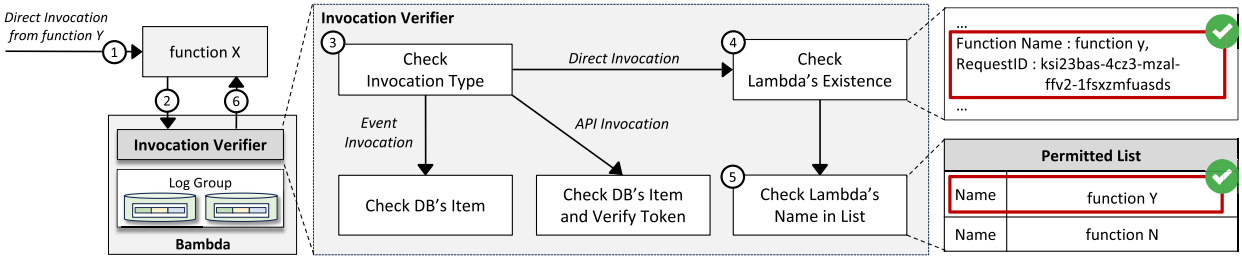


FIGURE 7. The steps for verifying function invocation using the Invocation Verifier, illustrating type-specific verification processes for direct calls, event triggers, and API requests.

explicitly listed, the verification results are sent to the Verifier (6), and the function call is approved. If an unauthorized function had attempted to invoke `function X`, the request would have been immediately blocked.

For the event-based function calls, verification focuses on analyzing the data structure and occurrence pattern of the event. The verification code first accesses the source resource to confirm that the event actually occurred. For example, in a DynamoDB stream event, the system verifies event data by checking the key structure of the original table and the data types of each field. Additionally, by monitoring the frequency of consecutive event occurrences, the system detects excessive updates in DynamoDB or unnecessary chained Lambda function calls. If suspicious patterns are identified, corresponding function calls are temporarily restricted to prevent potential misuse.

For the direct calls to external services, verification analyzes the origin and context of requests made from public IPs. The verification code examines HTTP header fields to ensure that the Origin and Referer values are consistent with legitimate web page requests. It also evaluates the frequency and pattern of requests originating from the same IP within a short time frame to identify potential anomalies. For requests to external services, the system verifies the security token embedded in the request payload, helping detect anomalous or spoofed requests. This approach ensures the authenticity of requests passing through the API Gateway, as the absence of a security token or a malformed structure may indicate spoofing attempts. Through multi-layered verification, the system proactively blocks malicious API calls and mitigates denial-of-service (DoS) attacks via public IPs.

The detailed invocation verification steps are explained in Algorithm 1. The verification process begins when a function call event is received; the system first records the function call event (line 1) and identifies the invocation type by analyzing the event payload or HTTP header information (line 2). Based on the identified invocation type, a tailored verification process is executed. When the invocation type is direct (line 3), the system first retrieves the corresponding allow list (line 4-5). Then, it verifies the caller function's existence via real-time Workflow Log Stream records (line 6). If the existence is confirmed, the system checks whether the caller is authorized by the allow list before granting or denying access to resources (lines 7-10).

For event-based invocations (line 11), the system fetches the originating item, verifies its conformity with the source schema (lines 12-13), and monitors invocation patterns to detect abnormal behaviors (lines 14-15). If any discrepancies or anomalies are found, the event is flagged and access is restricted. For external service requests (line 16), the system examines the embedded security token (line 17) and compares the payload structure against the expected schema to determine legitimacy (lines 18-20), thus verifying the authentication status of the request (lines 21-23). If all verification steps are successfully passed, function execution is allowed (line 24); otherwise, access is immediately blocked if any issues are detected during the verification process.

VI. EVALUATION

This section evaluates the security effectiveness and performance impact of BAMBDA in a real-world serverless environment. We evaluate the system's ability to detect and prevent unauthorized function invocations and measure the associated runtime overhead and cost impact.

A. IMPLEMENTATION

We implemented BAMBDA as a prototype for detecting and blocking unauthorized serverless function calls within the AWS platform. Our implementation is based on Node.js [34] and YAML. We deploy our solution on AWS Lambda and manage the serverless infrastructure using Serverless Framework v4.25. For centralized workflow monitoring, we use AWS CloudWatch to create a unified log group. The verification mechanisms are implemented using the AWS SDK library, which also manages all interactions with AWS services when deploying functions with verification code. We have implemented differentiated verification logic for each call type, with all verification results recorded in centralized log streams to provide comprehensive audit trails and support post-analysis.

B. TEST ENVIRONMENTS

The experiments were conducted in the ap-northeast-2 region of AWS Lambda. The AWS Lambda environment was configured with 128MB of memory allocation, and since AWS Lambda allocates CPU power in proportion to the amount of memory configured, we were provided with Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz processors.

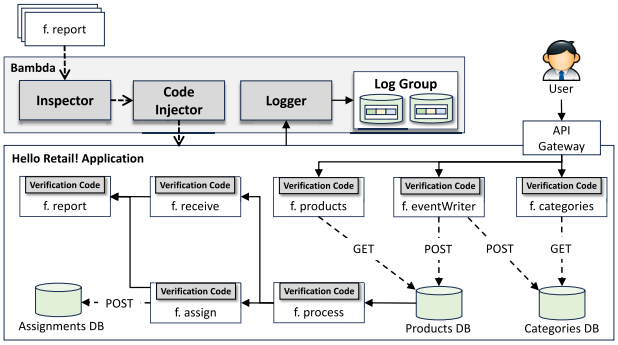


FIGURE 8. The testbed overview of the 'Hello Retail!' real application, comprising 7 serverless functions with integrated BAMBDA framework.

The maximum execution timeout for Lambda functions was set to the default of 3 seconds, and Node.js 20.x was used as the runtime for all deployed functions. During testing, we used the AWS CLI to invoke Lambda functions and test API Gateway endpoints.

Figure 8 illustrates the structure of our test application, Hello Retail! [35], with BAMBDA attached to the application. This application consists of seven independently deployed serverless functions built with Node.js, including step functions that automate function workflow connections. Hello Retail! provides e-commerce services such as product registration, search, and checkout. The application's architecture features complex interfunction call relationships and database interactions with DynamoDB, making it an ideal for comprehensive evaluation of direct function invocation, event-chained invocation, and external API requests via API Gateway.

We implemented three attack scenarios aligned with the threat model (C1-C3). In C1, the 'categories' function is modified to embed a call to 'eventWriter' function, enabling indirect privilege escalation and unauthorized data insertion. In C2, malformed data is written to Products DB via 'eventWriter' function, triggering process without validation and causing incorrect execution. In C3, a spoofed API request is sent to Products DB with manipulated headers, resulting in unauthorized access through API Gateway. These scenarios demonstrate how attackers can exploit legitimate invocation paths to compromise function workflows.

C. EFFECTIVENESS OF INVOCATION TYPE-SPECIFIC VERIFICATION

Based on the three attack scenarios described in the previous section (C1-C3), we evaluated the ability of BAMBDA to detect and block unauthorized function invocations in real-world AWS serverless environments. To conduct a comprehensive assessment, we designed and implemented three attack scenarios that exploit different vulnerabilities in AWS Lambda functions and evaluated the ability to detect and prevent such attacks. Figure 9 shows the implementation and results of the attack scenarios we successfully executed in practice. In the direct invocation attack(A-1), we exploited excessive IAM permissions by injecting malicious code into the 'categories'

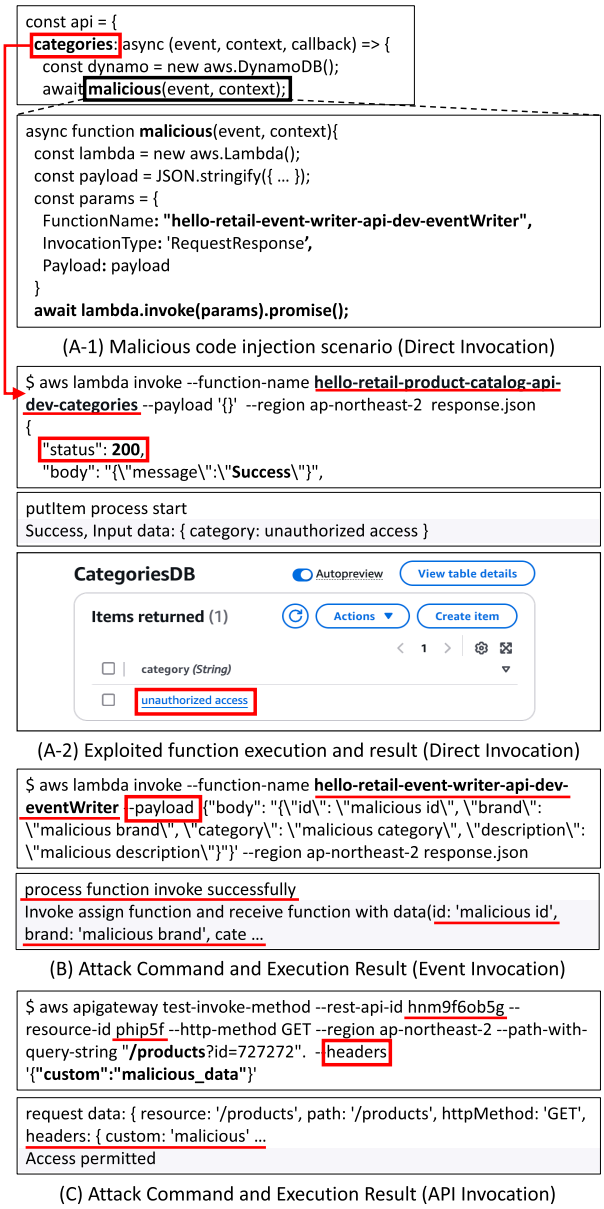
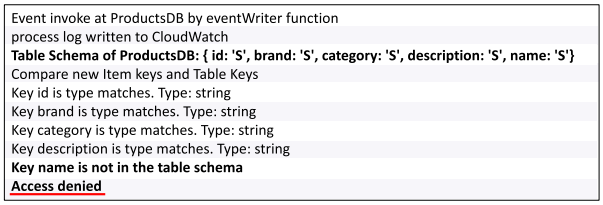


FIGURE 9. The results of preventing an unauthorized attack that exploits direct function invocations.

function. As shown in the code snippet, we created a malicious function named malicious within the 'categories' function that enabled the 'categories' function to invoke the 'eventWriter' function without proper authorization checks. When executed via the AWS CLI(A-2), this attack successfully returned a status 200 and added an unauthorized entry to the CategoriesDB, confirming that ''unauthorized access'' was indeed arbitrarily added to the CategoriesDB and demonstrating the vulnerability of traditional IAM permission models.

For the event chaining attack (B), we executed a command to invoke the 'eventWriter' function with malformed data by injecting anomalous entries into the Products DB. This event triggered the automatic invocation of the 'process' function, and we confirmed that the invocation was successful

(A) The verification result for Direct Invocation



(B) The verification result for Event Invocation

(C) The verification result for API Invocation

FIGURE 10. The results of unauthorized attacks using event function invocations.

without proper validation, potentially forcing functions to process malicious data. In the API Invocation attack (C), we sent a spoofed HTTP request with a malicious custom header to the API Gateway endpoint connected to the 'products' function. This attack also resulted in ''access permitted'' with no blocking, demonstrating how attackers can exploit weaknesses in the security of API Gateway by manipulating request parameters and headers, allowing unauthorized access to sensitive data or function execution. To defend against these successful attacks, we attached BAMBDA into the application, with the verification results indicated in Figure 10.

1) DIRECT INVOCATION

As shown in Figure 10(A), when the 'categories' function attempts to invoke the 'eventWriter' function, the verification code first logs ''lambda is invoked by 'categories' function'' to the Log group and executes logging and verification in parallel. It then searches for logs related to the 'categories' function with a specific timestamp and extracts the function's log stream name and function name. The verification code compares the extracted function name to the permitted list authorized to invoke 'eventWriter' function. Since the 'categories' function is only designed for GET operations and is not in the permitted list for invoking 'eventWriter' function (which performs POST operations), the verification code identifies this as an unauthorized flow and issues an ''Access denied'' decision. Access is immediately denied before any database operations can occur, effectively preventing the privilege abuse attack.

2) EVENT CHAINING INVOCATION

As depicted in Figure 10(B), when the 'eventWriter' function triggers an event that activates the 'process' function, the verification code first records ''Event invoke at Products DB by 'eventWriter' function''. The system then performs a schema validation by comparing the structure of the newly inserted item with the expected Products DB table schema. The verification code checks each key field: verifying that id is a string type, brand matches the expected string type, category is formatted correctly, and so on. In this case, the system detects that the key name is missing from the table schema, thereby catching a potential data integrity problem. This results in an ''Access denied'' decision, preventing the 'process' function from executing with potentially malicious data and ensuring that it cannot be used as a trigger mechanism for further exploit chains.

3) API INVOCATION

As illustrated in Figure 10(C), when a suspicious request reaches the 'products' function, the verification code logs ''Invocation from account ID: 12345678912''. The system then initiates a comprehensive verification of both the request headers and the data payload. First, it checks for the presence of a security token that should be included in legitimate requests. The verification code analyzes the request data structure, examining the path ('/products'), the HTTP method ('GET'), and the headers, including the suspicious custom header containing 'malicious' values. During this analysis, the system detects ''No security token in data'' and immediately denies access to the function. This security measure effectively prevents attackers from exploiting shared IP vulnerabilities by ensuring that all requests contain proper authentication tokens and originate from legitimate sources.

D. PERFORMANCE OVERHEADS

In this experiment, we quantitatively evaluate the impact of integrating BAMBDA on the execution performance of serverless functions. We evaluated all seven functions of the target application by measuring them under both cold start [36] and warm start [37] conditions using the AWS CLI [38]. For the cold start measurements, we executed each function 50 times, while for the warm start measurements, we executed each function 100 consecutive times and calculated the average execution time.

As shown in Figure 11(A), the cold start duration increases by approximately 4-58 ms for all functions. Notably, 'categories' function shows the smallest absolute increase of only about 4.23 ms, which is roughly 0.58% over its baseline. In contrast, the 'eventWriter' function incurs the largest overhead, with an increase in execution time of 57.67 ms (8.28%), primarily due to its frequent interactions with external services. These results suggest that, under cold start conditions, the overall overhead is more influenced by the inherent provisioning time of serverless containers than by the initialization of the injected verification logic.

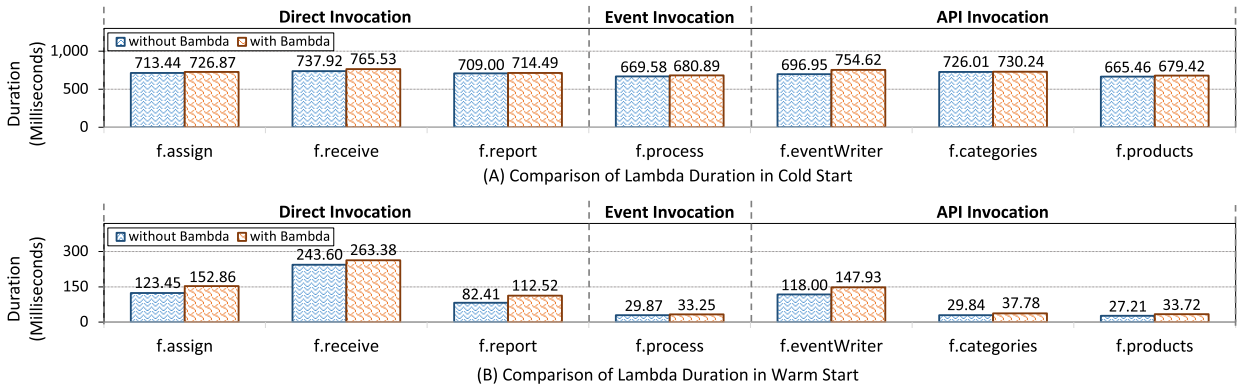


FIGURE 11. The results of duration normalization across the applications.

TABLE 2. Notations and units of cost model.

Symbol	Description	Unit
C_{comp}	Compute Cost	USD
C_{total}	Total Cost	USD
C_{inv}	Invocation Cost	USD
M_{size}	Memory Size	MB
D_{ms}	Duration	ms
$C_{\text{GB-ms}}$	Cost per GB-ms	USD/GB-ms

The result for warm starts is shown in Figure 11(B). Among all functions, the 'process' function shows the smallest overhead, increasing by approximately 3.38 ms (11.32%), while the 'report' function shows the largest increase of 30.11 ms (36.55%). These results suggest that the verification of direct invocations or database manipulation functions (e.g., 'assign') may involve additional interactions with external services, thus incurring more overhead.

The findings show that in cold starts, due to function initialization and container provisioning, latency is generally higher compared to warm starts. Overall, using BAMBDA results in approximately 4.02% overhead for cold starts and 20.97% for warm starts. Especially, functions of the Direct Invocation type that require more complex interactions with external services tend to have higher overhead; however, given the trade-off between improving security and performance, this level of overhead is considered acceptable. While the relative percentages may appear somewhat high, the absolute increase is in the tens of milliseconds range, suggesting that the impact on user experience is negligible in real-world. Furthermore, the verification logic is function-specific and does not depend on the total number of deployed functions or the application's overall scale, thereby ensuring that the system remains scalable even as the workload increases.

E. ADDITIONAL COST OVERHEADS

We analyzed the cost implications of applying BAMBDA to AWS Lambda environments. Figure 12 illustrates the costs based on latency measurements taken in Section VI-D under warm start conditions. AWS Lambda costs are determined by three key factors: function execution time, allocated memory, and invocation frequency. The implementation of BAMBDA introduces additional costs compared to standard function execution due to its security validation mechanisms and centralized logging in CloudWatch. AWS Lambda costs are proportional to both request-based charges and execution time, with AWS Lambda charging $0.2 USD per million requests and calculating execution costs at $0.00001333 USD per second for 1 GB of memory. Table 2 defines the relevant variables, with the total Lambda cost calculated using the following equations:

$$C_{\text{comp}} = \left(\frac{M_{\text{size}}}{1024}\right) \times D_{\text{ms}} \times C_{\text{GB-ms}} \quad \text{(a)}$$

$$C_{\text{total}} = C_{\text{inv}} + C_{\text{comp}} \quad \text{(b)}$$

Figure 12 shows the changes in total costs for AWS Lambda functions. Our analysis showed that the 'assign' function experienced the highest cost increase at 21.24%, while the 'categories' and 'process' functions experienced increases of 17.7% and 7.5%, respectively. This significant variation in cost impact is due to differences in how each call method interacts with our security validation layer. Functions with more complex security validation requirements naturally incur higher additional execution times, resulting in proportionally higher costs. Across all functions tested, we observed an average cost increase of 20.17%.

Ultimately, while our approach increases costs compared to a baseline with no security measures, the costs associated with security breaches in serverless environments can far outweigh these operational increases. Also, for organizations with limited security budgets that need to make a reasonable compromise, BAMBDA enables teams to maintain an adequate security posture with minimal additional effort by supporting function call validation while maintaining operational balance.

VII. LIMITATION AND DISCUSSION

As with other research, the current system is subject to several limitations that require further research. In this section, we undertake a review of these constraints and propose a

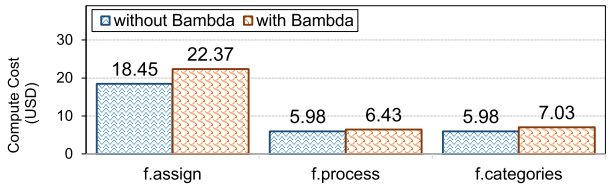


FIGURE 12. Additional compute costs for functions from each invocation type with and without the BAMBDA applied.

number of improvements with the aim of extending the capabilities of BAMBDA across various dimensions.

A. EXTENSIBILITY IN PROGRAMMING LANGUAGES AND CLOUD PLATFORMS

AWS Lambda supports various serverless runtime environments, including Node.js [34], Python [39], and Java [40]. However, the current implementation of BAMBDA operates exclusively with Node.js functions within the Serverless Framework, leveraging code reconstruction and multi-stage verification. This limitation presents two key challenges.

First, the restricted support for programming languages may hinder the scalability and security consistency of our system in heterogeneous serverless environments. Addressing this issue requires developing a universal verification mechanism that functions across diverse programming languages. Our approach involves designing separate adapters or a shared validation component that accommodates different language-specific runtime structures. As future work, we plan to enhance BAMBDA with LLM-based behavior modeling, least-privilege policy extraction, and real-time detection of inappropriate behaviors, along with broader compatibility across programming languages and cloud platforms, enabling adaptive verification and wider applicability.

Second, cloud platform dependency presents another limitation. While our system is designed for seamless integration with AWS Lambda, it is not currently compatible with other cloud providers such as Google Cloud Functions or Azure Functions. Given the increasing prevalence of multi-cloud deployments, we also plan to support other vendors in the future. However, achieving cross-platform compatibility is non-trivial due to fundamental differences in runtime execution models, event dispatch mechanisms, and logging APIs across platforms. These differences make it difficult to reuse our current implementation as-is and would require substantial redesign of the verification logic. Expanding BAMBDA into a multi-cloud environment requires establishing a standardized execution environment and a unified interface across serverless platforms. This extension will enable our system to provide security verification capabilities without being restricted to a specific cloud provider.

B. SERVICE-BASED INVOCATION VERIFICATION

AWS Lambda enables event-driven function execution by integrating with various AWS services, such as S3 [41], SQS [42], and SNS [43]. However, BAMBDA currently enforces security verification and policy enforcement only for events triggered by DynamoDB, S3, and API Gateway. This limitation restricts its functional scalability and compatibility, as AWS Lambda can be invoked by numerous other service events that our system does not yet support. To address this, we plan to expand our system's validation scope by developing service-specific verification mechanisms that process and validate events from a broader range of AWS services.

Additionally, our system currently performs security verification on a single-event basis, limiting its ability to analyze multi-service event chains. Beyond single-event verification, establishing a mechanism for securing inter-service event chains is essential. For instance, consider a scenario where an object uploaded to S3 triggers an SNS notification, which subsequently invokes a Lambda function. In this case, our system verifies the SNS-triggered event but does not analyze the preceding S3 event, creating a security gap. To address this, we propose leveraging distributed tracing techniques to track event flows across multiple AWS services and assess security states at each transition point. Specifically, we aim to implement an event-labeling mechanism that enables our system to systematically verify the integrity and security of multi-service event chains, ensuring comprehensive security enforcement.

C. POISONED DATA VALIDATION

BAMBDA performs data validation using a schema comparison approach. While effective in verifying data format and structure, this method has inherent limitations in assessing data integrity and semantic validity. To overcome these limitations, additional validation mechanisms are required. We propose two major enhancements to improve the robustness of data validation. First, we plan to incorporate content-based validation, which not only ensures structural conformity but also evaluates the validity of actual data values. This will enable the system to detect and block falsified or manipulated data.

Second, we intend to implement a cross-referencing validation mechanism that compares incoming data against pre-existing records in a database or external trusted sources. This approach will enhance data reliability and effectively mitigate attacks that exploit disguised or misleading data. By integrating these enhancements, our system aims to strengthen data integrity verification, providing a more comprehensive defense against data tampering and unauthorized modifications in serverless computing environments.

D. OPERATIONAL RISKS OF AUTOMATED CODE INJECTION

BAMBDA is designed to automatically inject security verification logic at the entry point of each function without interfering with the original business logic. However, the injection process itself expands the security surface, thus requiring careful consideration from management, operational, and compliance perspectives. While our experiments did not encounter such issues, external conditions such as

log collection failures or invocation delays may affect the entire function flow through the injected module. Moreover, in highly regulated environments, the mere fact that deployed code is modified without explicit developer approval may raise concerns about deployment transparency.

BAMBDA also relies on a centralized logging system to trace and verify inter-function invocation relationships. This is a critical element for real-time security decision-making; however, in high-traffic environments, the logging infrastructure may become a bottleneck. In our implementation, we created independent log streams in CloudWatch to minimize performance impact, but in environments with extremely high invocation volumes, log collection delays or processing limitations may still arise. Nevertheless, due to the nature of serverless architectures—where each function scales and executes independently—the logging load tends to distribute proportionally across the application. Furthermore, cloud-native logging services such as AWS CloudWatch are designed to meet such scalability demands.

To mitigate these risks, we suggest applying formal verification to the injected logic and incorporating optional developer approval workflows in the CI/CD pipeline. There is also the potential that the injected module itself could become a target of attack. If the module is compromised, the entire application could be affected. Therefore, we recommend executing the verification module under a least-privilege IAM role and restricting any external network access.

E. QUANTITATIVE EVALUATION OF VERIFICATION ROBUSTNESS

BAMBDA is structurally designed for secure operation, but quantitative evaluation of the verification logic's detection accuracy may also be important. Since BAMBDA generates execution logs at every function invocation and verifies call relationships based on them, the likelihood of false positives is extremely low. In particular, because the system only blocks unauthorized invocations and automatically allows those listed in the allow list, no false positives were observed during our experiments.

However, in exceptional cases—such as missing security tokens in external requests or temporary delays in log transmission—there is a possibility of false negatives. These are not due to logical flaws in BAMBDA itself, but rather stem from cloud vendor-dependent execution conditions, such as delayed log delivery or non-standard event structures. Nevertheless, to quantitatively analyze even these rare possibilities, we plan to evaluate detection precision using diverse datasets that include both benign and anomalous scenarios. Metrics such as precision, recall, and F1-score will be used in future work.

Additionally, the system's resilience to potential evasion tactics—such as an attacker attempting to bypass security verification—is another critical aspect of evaluation. For example, attacks may attempt to forge caller information or mimic authorized call paths to evade detection. Currently, BAMBDA is designed to effectively prevent such bypass attempts by leveraging a log-based tracking mechanism that traces inter-function relationships. However, we plan to quantitatively evaluate its resilience through tests involving a wide range of evasion scenarios in future research.

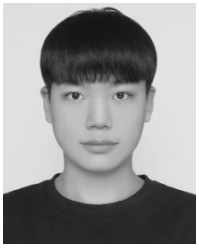
VIII. CONCLUSION

In this paper, we proposed BAMBDA, a dynamic framework for detecting and blocking unauthorized attacks that exploit serverless function invocation conditions and IAM configurations in real-world serverless computing environments. Specifically, our system not only prevents abnormal access attempts via indirect function invocation but also records blocked function workflows, enabling developers to proactively mitigate potential security vulnerabilities. Through comprehensive attack scenario testing, we demonstrated that our system effectively detects and blocks all attack attempts while maintaining reasonable performance overhead. These results confirm that our system provides significant security benefits while serving as a practical reference for strengthening serverless security. Furthermore, our system offers new insights into security management in environments where dynamic verification based on invocation conditions remains challenging.

REFERENCES

[1] DATADOG. (2025). *The State of Serverless*. [Online]. Available: https://www.datadoghq.com/state-of-serverless
[2] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Trans. Services Comput.*, vol. 16, no. 2, pp. 1522–1539, Mar. 2023.
[3] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–32, Jan. 2022.
[4] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–34, Jan. 2022.
[5] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," *J. Cloud Comput.*, vol. 10, no. 1, pp. 1–29, Jul. 2021.
[6] S. Eismann, J. Scheuner, E. v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Trans. Softw. Eng.*, vol. 48, no. 10, pp. 4152–4166, Oct. 2022.
[7] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Softw.*, vol. 38, no. 1, pp. 32–39, Jan. 2021.
[8] M. Wu, Z. Mi, and Y. Xia, "A survey on serverless computing and its implications for JointCloud computing," in *Proc. IEEE Int. Conf. Joint Cloud Comput.*, Aug. 2020, pp. 94–101.
[9] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proc. Int. Conf. Internet Things Design Implement.*, Apr. 2019, pp. 225–236.
[10] J. Xiong, M. Wei, Z. Lu, and Y. Liu, "Warmonger: Inflicting denial-of-service via serverless functions in the cloud," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 955–969.
[11] D. Kelly, F. G. Glavin, and E. Barrett, "Denial of wallet—Defining a looming threat to serverless computing," *J. Inf. Secur. Appl.*, vol. 60, Jan. 2021, Art. no. 102843.
[12] DivvyCloud. (2021). *2020 Cloud Misconfiguration Report*. [Online]. Available: https://www.bankinfosecurity.com/whitepapers/2020-cloud-misconfigurations-report-w-6009
[13] Google Cloud. (2025). *Threat Horizons*. [Online]. Available: https://services.google.com/fh/files/misc/threat_horizons_report_h2_2024.pdf
[14] AWS. (2025). *AWS IAM Access Analyzer*. [Online]. Available: https://aws.amazon.com/iam/access-analyzer/
[15] G. Hu, Y. Wu, G. Chen, T. T. A. Dinh, and B. C. Ooi, "SeSeMI: Secure serverless model inference on sensitive data," 2024, *arXiv:2412.11640*.

[16] I. Polinsky, P. Datta, A. Bates, and W. Enck, "GRASP: Hardening serverless applications through graph reachability analysis of security policies," in *Proc. ACM Web Conf.*, May 2024, pp. 1644–1655.
[17] D. S. Jegan, L. Wang, S. Bhagat, and M. Swift, "Guarding serverless applications with Kalium," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 4087–4104.
[18] I. Polinsky, P. Datta, A. Bates, and W. Enck, "SCIFFS: Enabling secure third-party security analytics using serverless computing," in *Proc. 26th ACM Symp. Access Control Models Technol.*, Jun. 2021, pp. 175–186.
[19] D. Sirone Jegan, L. Wang, S. Bhagat, T. Ristenpart, and M. Swift, "Guarding serverless applications with SecLambda," 2020, *arXiv:2011.05322*.
[20] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates, "Valve: Securing function workflows on serverless computing platforms," in *Proc. Web Conf.*, Apr. 2020, pp. 939–950.
[21] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein, "Secure serverless computing using dynamic information flow control," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–26, Oct. 2018.
[22] AWS. (2025). *AWS Lambda*. [Online]. Available: https://aws.amazon.com/ko/lambda/
[23] Google Cloud. (2025). *Cloud Run Functions*. [Online]. Available: https://cloud.google.com/functions
[24] Azure. (2025). *Azure Functions*. [Online]. Available: https://azure.microsoft.com/ko-kr/products/functions
[25] sysdig. (2025). *Sysdig 2024 Cloud-native Security and Usage Report*. [Online]. Available: https://sysdig.com/2024-cloud-native-security-and-usage-report/
[26] J. Xiong, M. Wei, Z. Lu, and Y. Liu, "Warmonger attack: A novel attack vector in serverless computing," *IEEE/ACM Trans. Netw.*, pp. 1–16, Jan. 2024.
[27] J. Wen, Z. Chen, F. Sarro, Z. Zhu, Y. Liu, H. Ping, and S. Wang, "LLM-based misconfiguration detection for AWS serverless computing," 2024, *arXiv:2411.00642*.
[28] P. Padma and S. Srinivasan, "Dauth—Delegated authorization framework for secured serverless cloud computing," *Wireless Pers. Commun.*, vol. 129, no. 3, pp. 1563–1583, 2023.
[29] P. Gupta, A. Moghimi, D. Sisodraker, M. Shahrad, and A. Mehta, "Growlithe: A developer-centric compliance tool for serverless applications," in *Proc. IEEE Symp. Secur. Privacy (SP)*, Dec. 2024, p. 99.
[30] A. Kumari, Md. Akram Khan, and B. Sahoo, "Workflow sensitive access management in serverless computing," in *Proc. IEEE 2nd Int. Symp. Sustain. Energy, Signal Process. Cyber Secur. (iSSSC)*, Dec. 2022, pp. 1–6.
[31] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "ALASTOR: Reconstructing the provenance of serverless intrusions," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 2443–2460.
[32] A. Ashkenazi, E. Grolman, A. Elyashar, D. Mimran, O. Brodt, Y. Elovici, and A. Shabtai, "SMART: Serverless module analysis and recognition technique for managed applications," in *Proc. IEEE 24th Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, May 2024, pp. 442–452.
[33] AWS. (2025). *Amazon Cloudwatch*. [Online]. Available: https://aws.amazon.com/cloudwatch/
[34] Nodes. (2025). *Run Javascript Everywhere*. [Online]. Available: https://nodejs.org/
[35] S. Eismann. (2021). *Hello Retail!s Application*. [Online]. Available: https://github.com/SimonEismann/hello-retail
[36] S. Ristov, C. Hollaus, and M. Hautz, "Colder than the warm start and warmer than the cold start! Experience the spawn start in FaaS providers," in *Proc. Workshop Adv. tools, Program. Lang., PLatforms Implementing Evaluating algorithms Distrib. Syst.*, Jul. 2022, pp. 35–39.
[37] J. Carreira, S. Kohli, R. Bruno, and P. Fonseca, "From warm to hot starts: Leveraging runtimes for the serverless era," in *Proc. Workshop Hot Topics Operating Syst.*, Jun. 2021, pp. 58–64.
[38] AWS. (2025). *AWS Command Line Interface*. [Online]. Available: https://aws.amazon.com/cli/
[39] (2025). *Python*. [Online]. Available: https://www.python.org/
[40] Java. (2025). *Get Java for Desktop Aplications*. [Online]. Available: https://www.java.com/
[41] AWS. (2025). *Amazon S3*. [Online]. Available: https://aws.amazon.com/ko/s3/
[42] AWS. (2025). *Amazon Simple Queue Service*. [Online]. Available: https://aws.amazon.com/sqs/
[43] AWS. (2025). *Amazon Simple Notification Service*. [Online]. Available: https://aws.amazon.com/sns/



CHANGHEE SHIN is currently pursuing the B.S. degree with the Department of Computer Science and Engineering, Incheon National University. His research interest includes serverless computing security.

BOM KIM is currently pursuing the M.S. degree with the Department of Computer Science and Engineering, Incheon National University. Her research interests include the automation of intent-driven security policy generation and validation for advanced cloud-native environments.

SEUNGSOO LEE received the B.S. degree in computer science from Soongsil University, the M.S. degree in information security from KAIST, and the Ph.D. degree in information security from KAIST, in 2020. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Incheon National University. His research interests include developing secure and robust cloud/network systems against potential threats.