

Detailed Security Analysis of Serverless Functions with Interpreted Languages

Team 26

Client and Advisor: Dr. Berk Gulmezoglu

Michael Gohr

Dillon Hacker

Cameron Hurt

Samuel Potter

Trent Walraven

sdmay24-26@iastate.edu

<https://sdmay24-26.sd.ece.iastate.edu/>

Introduction/Background

1.1 PROBLEM STATEMENT

This project is dedicated to conducting a comprehensive security analysis of serverless functions executed in interpreted languages, a prevalent feature across various cloud platforms. These platforms enable users to run code seamlessly without managing server operations. However, this convenience potentially compromises security, as customers typically operate under the assumption that their functions are isolated from others sharing the same physical infrastructure. The research aimed to challenge this assumption by developing a side channel attack that can differentiate and potentially expose the contents of one function from another on the same server. This poses a significant risk to the presumed confidentiality and security of cloud computing environments, where customers expect their operational data to remain private and unknown to others on the same platform.

INTENDED USERS AND USES

- Researchers
- Hackers
- Students studying computer science, computer engineering, or cybersecurity engineering
- Firecracker developers
- Programmers
- Large scale users of serverless functions, like businesses, should understand the potential dangers that come with running these serverless functions in a public cloud, and the potential for them to leak data by being co-located with a malicious actor.

Prior Work/Solutions

There has been nothing done that is exactly like this before. Some things cover partial areas of what we are doing, but did not go this far. For example this talk “Peeking Behind Serverless Functions” <https://www.usenix.org/system/files/conference/atc18/atc18-wang-liang.pdf>, talks about finding co-residency with your own serverless functions. Along with there being various attacks designed to leak data from processors like a cache timing attack seen here <https://cr.yp.to/antiforgery/cachetiming-20050414.pdf>. Our project will be some combination of both of these, with the goal requiring us to combine the ideas of finding if we are running alongside another serverless function, and then attempting to leak data from it.

1.2 REQUIREMENTS

- Resource Requirements
 - The code that we create as part of the project itself
 - The serverless function SDKs for intended cloud providers
 - A server, for use in local testing and execution without the dangers of a public cloud, set up with the help of our client
 - A Firecracker and Docker environment that we can use for testing and data collection of these functions
- Legal Requirements
 - If a vulnerability is found, safely and securely report it through the proper channels, and do not disclose any information until

- Cannot interact (leak, break, stop, etc), accidentally or intentionally, code that we do not have explicit ownership and control of while any attack code is running.
- Performance Requirements
 - Needs to fit in the memory, and time limit of serverless functions
 - Need to ensure that our time measurement programs are running on a single computer core
 - Output sufficient amount of data that can train a learning model during initial testing
 - Reduce the output data once a classification model exists to optimize the time and space required.
- Maintainability Requirements
 - Write code that uses the basic functions of the language to allow high likelihood for cross-provider interoperability
 - Include verbose comments about how the code is functioning the usage of SDK specific instructions that are not 1 for 1 across providers
 - General code development practices that allow for continued maintaining and usage of code by others in the future.
- Testing Requirements
 - Environment we set up must mimic actual serverless function runners such as AWS Lambda
 - What data can be gathered as code is running, and how is it useable
 - That the collected data remains consistent across runnings of the code showing patterns that can be trained on
 - Methods to effectively transmit, organize, store, and classify data continue to work for the collected data, and remain consistent
 - We will have to ensure that each of our programs is running as efficiently as possible and outputting accurate data
 - We will have to consistently test the mimicked serverless environment and ensure that our code is working properly in this environment

1.3 ENGINEERING STANDARDS

Wonderless:

- Dataset of real-world serverless applications
- Created by trawling GitHub for serverless.yml configuration files
- Broad overview of what serverless applications are like in reality
- See paper [here](#)

AWS- Lambda

- Service offered by AWS - serverless computing
- Linked GitHub with AWS containers that are similar to the real Lambda environment
- Standard across all serverless functions provided by AWS

Serverless Framework

- Commonly used framework across serverless computing vendors

- Tied to AWS Lambda (development and deployment framework for it)
- Managing serverless functions one higher level of abstraction

General Cybersecurity Research Standards

- Ethics
- Responsible disclosure
- Research transparency
- Data protection and privacy

Security concerns and countermeasures

Through research, several security vulnerabilities inherent to the architecture of serverless computing have been identified, particularly when functions are executed in shared environments using interpreted languages. The primary concerns involve:

Data Leakage: Findings demonstrate potential scenarios where sensitive data could be inadvertently exposed between co-located lambda functions. This leakage primarily occurs through side channels, where one function can infer data about another based on resource usage patterns (e.g., CPU cache, execution time).

Cross-Function Interference: Functions running in close proximity may interfere with each other's performance and operations, leading to service degradation or unintended behaviors, which can be exploited maliciously.

Insufficient Isolation: The standard isolation mechanisms employed by serverless platforms may not fully prevent the leakage of low-level, hardware-based side-channel signals, thus posing a risk of cross-function espionage.

Proposed Countermeasures

To mitigate these security risks, we propose several countermeasures and best practices:

Enhanced Isolation Protocols: Cloud providers should implement more robust isolation mechanisms that extend beyond the software layer, incorporating hardware-assisted solutions to segregate the execution traces of co-located functions.

Regular Audits and Monitoring: Implement continuous monitoring tools that can detect anomalous behaviors indicative of side channel attacks. Regular security audits can help identify and mitigate newly emerging vulnerabilities in serverless architectures.

Resource Usage Limits: By setting stringent resource usage limits, cloud providers can reduce the risk of one function impacting another's execution environment significantly enough to extract meaningful data.

Developer Education and Guidelines: Educate developers on best practices for designing serverless applications, emphasizing secure coding practices that minimize the risk of introducing side channel vulnerabilities.

The implementation of these countermeasures requires collaboration between cloud providers, security researchers, and the broader developer community. Cloud providers must lead with innovative isolation technologies and robust monitoring systems, while developers must adhere to security best practices in their application designs.

Description of how your design has evolved since 491

The design has evolved significantly since 491. To narrow the scope of the project, the group focused on writing the attack code in one language instead of four or five to focus on working together through problems instead of splitting up to solve the same problems. This change was beneficial to the group because all members were working on the same code instead of separate code pieces. Another change the group made was to not utilize a learning model at this final stage of development. Despite the utility that a

learning model could have provided, the group focused on finding examples of lambdas that could be profiled with the human eye. This put some constraints on the project, as there were many lambdas that can not be differentiated by human sight alone, but it allowed for more time to fine tune and streamline much of the project. These tradeoffs ended up benefiting the progress of the project.

2 Design

2.1 DESIGN CONTENT

The project is designed to investigate potential data leakage in serverless functions executed in interpreted languages, with a focus on the risks associated with shared server environments. The methodology centers on two primary phases:

Detection of Shared Server Environments: The process begins by establishing whether a function is co-located with others on the same physical server. This is achieved through the analysis of cache footprint patterns and timing information, which can indicate the presence of other active functions. A series of specially crafted diagnostic functions are used to measure anomalies and patterns consistent with shared usage.

Data Extraction Trials: Once a shared environment is identified, the next step involves attempting data extraction from the co-located functions. This phase utilizes side channel attacks, exploiting specific vulnerabilities found in the implementation of the serverless architecture, such as timing attacks and cache state analysis. The approach involves creating controlled, benign functions that simulate attack scenarios to understand how data might be inadvertently exposed through normal operation in a real-world environment.

Throughout the project, a combination of custom-built tools and existing software is utilized to simulate and monitor these environments accurately. The test setup mirrors a typical cloud service provider's environment (Amazon) to ensure that the findings are applicable and relevant to real-world scenarios.

2.2 Design Complexity

This project combines different computer engineering and computer science theories to form a unique challenge. There were several constraints put forth in this project that helped narrow down the scope, but left us with little wiggle room.

1. **Memory Caching** - In an effort to potentially gather information in our attack, an understanding of how memory caching works and the different algorithms a system could potentially implement. These algorithms include: least recently used, first-in first-out, etc..
2. **Programming** - Low level programming languages have the ability to communicate with the hardware level of a computer system, but not every system will accept their usage. We had to find suitable programming languages that could work with Firecracker, but also give us the ability to gather information about the hardware. Some examples include: Python, Rust, Go, etc...
3. **Cloud and Serverless Functions** - Creating code that reliably and efficiently runs in a cloud environment as a whole without incurring extreme cost requires engineering a solution that fits the scale required. This is further increased in serverless functions where the pricing is based on the

milliseconds that the code spends running, requiring highly optimized code to adequately prevent extreme cost.

4. Statistical Analysis - The attack we are attempting to use will require analysis of the timing it takes to gather data from each cache location. We know that each layer in the cache will take, X, amount of time, so this will tell us when we have cache misses in memory. Using this information we will create a chart that will reflect the cache misses. Time will be reflected with varying shades of color, then this visualization will be compared to a main database for any matches of known graphs.

2.3 Modern Engineering Tools

Server: To emulate the cloud environment in which a serverless function would run on, we needed to have a server set up. The only consideration for this was that the server was able to support the Firecracker software.

Firecracker: This was one of the constraints to our project, so we needed to use this service. Firecracker is a Kernel-based Virtual Machine which allows for fast execution speed. The role for this technology in our project is to support Lambda, which runs serverless functions for clients.

Docker: In the AWS Lambda environment, to further isolate users functions from each other, after placing them in a microVM, they are then placed into an undisclosed containerization platform. Containerization is a common practice to improve isolation, and we are using the most common Docker, as the location in which we will be running the Lambda execution environment inside the Firecracker microVMs. This is to as closely as possible replicate the real AWS Lambda environment.

AWS Lambda: Lambda is a Software as a Service provided by Amazon Web Services. Lambda is used to execute code quickly and efficiently via the cloud. This allows users to use the cloud's resources rather than their own.

Programming Languages: Python, Go, Ruby, and Perl will be used in many aspects of this project. We used these languages to determine the size of the system's L1, L2, and L3 cache. The programming languages we decided to use had to be compatible with AWS Lambda and we have the tools necessary to successfully execute a side channel attack. We used Python to create our attack code and it was successful.

2.4 DESIGN CONTEXT

The project will have an effect on all companies and users of serverless functions and specifically the software Firecracker. This will help the companies understand the potential issues that could come out of serverless functions, like potential vulnerabilities or data leakages.

List relevant considerations related to your project in each of the following areas:

Area	Description	Examples
Public health, safety, and welfare	This could affect the general safety of the data that users of serverless functions expect to have	If we leak information from other functions, then this could also be happening in the cloud today, and leaking private information from production code.

Global, cultural, and social	Not applicable	Not applicable
Environmental	Not applicable	Not applicable
Economic	This could have an economic impact on users of serverless functions. If there is an issue that makes them no longer feel safe enough to use these functions, then they could have added the cost of running locally instead of in the cloud.	If a vulnerability is found, and a company thinks that their code is longer safe in the cloud, they would go and buy servers, and run it themselves out of their own data center adding cost.

2.5 DESIGN DECISIONS

Several key decisions have been made to ensure the project runs as it would in the real world. One crucial decision is the use of a local server for testing functions, instead of the cloud, to prevent potential data leakage and to avoid cloud costs. This choice not only saves time and money but also eliminates the risk of exposing uncontrolled data.

Furthermore, the local server is configured to mimic a cloud environment, specifically mirroring setups similar to those of AWS. This configuration allows code to run in a way that would not require modifications when transitioning to a real-world environment.

Another constraint pertains to the nature of interacting with serverless functions. These functions are subject to highly limited runtime and memory constraints. Consequently, the attack code developed must be efficient enough to operate within these limits, typically requiring runtimes of less than a few minutes and minimal RAM usage to avoid automatic termination.

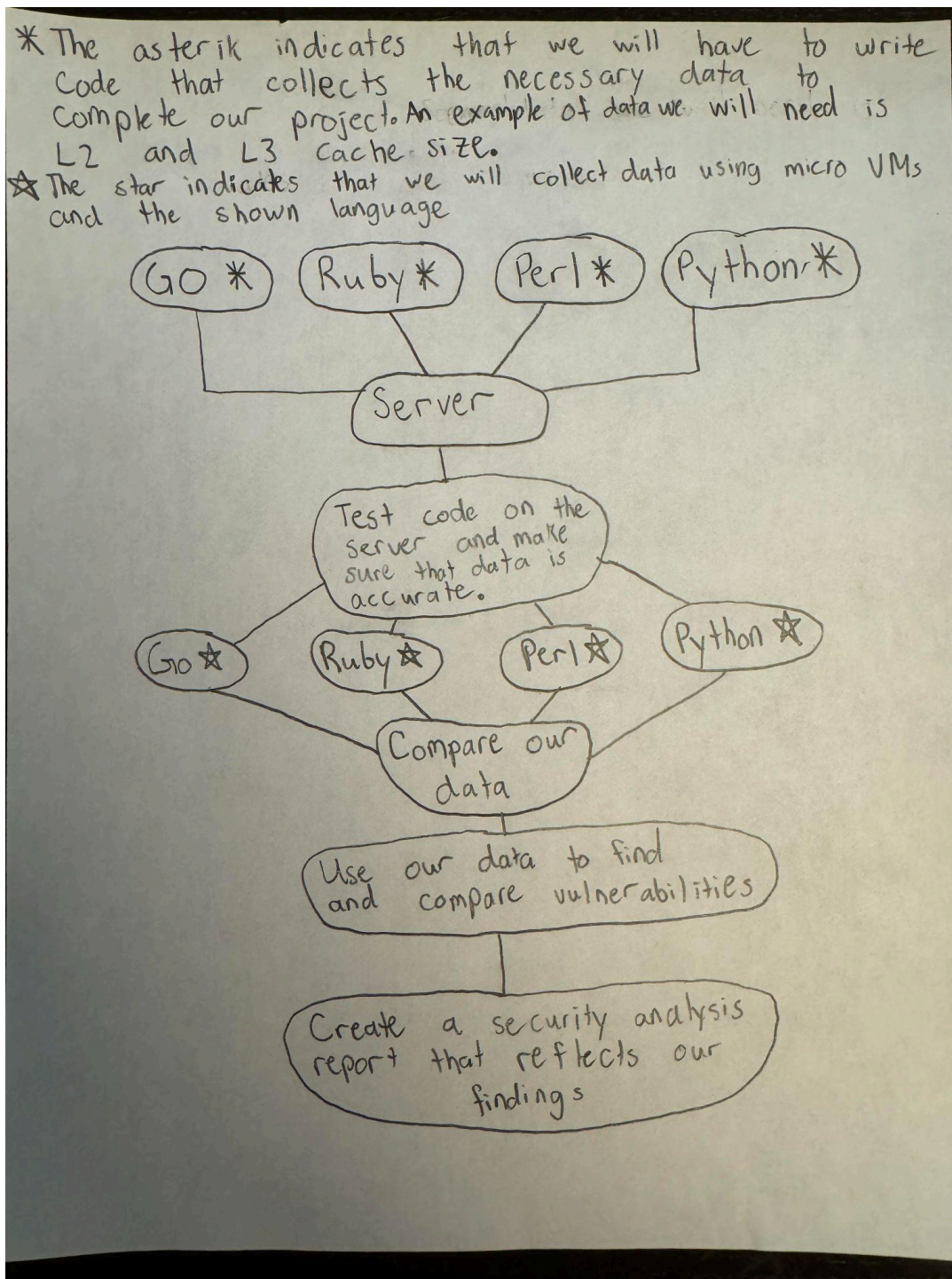
2.6 PROPOSED DESIGN

Multiple languages available in serverless functions are being used to explore the capabilities within each. Efforts include conducting cache timing and throughput tests outside the testing environment. These implementations are then transitioned into a Docker container that replicates the lambda execution environment to ensure they continue to function properly.

Following this phase, the focus will shift to scaling up data collection. The data gathered will be utilized to identify specific functions, further advancing the project's objectives.

2.7 Design o (Initial Design)

Design Visual and Description



The diagram outlines the workflow of a project that leverages serverless functions, highlighting the absence of physical hardware in its implementation. This approach focuses on using interpreted languages—Go, Perl, Python, and Ruby—to gather data about server performance and behavior. The collected data is then

tested on a server configured with numerous micro VMs, facilitating a detailed comparison of data and vulnerabilities across different interpreted languages.

Following the testing phase, the project culminates in the development of a security analysis report. This document will detail the findings in a manner accessible to individuals with a background in computer science or engineering, providing insights into language-specific vulnerabilities within serverless environments.

3 Testing

This project was predominantly software-oriented, with our testing efforts centered on the development and validation of specialized attack code. The primary objective was to devise a method that could precisely identify specific AWS Lambda functions by analyzing the distinct patterns they leave in the system's cache. This approach not only aimed to demonstrate the feasibility of such identification but also to expose potential vulnerabilities within serverless architectures.

3.1 UNIT TESTING

Only specific programming languages are permitted in serverless environments, which required us to source libraries compatible with the languages we implemented, enabling us to conduct tests locally without relying on a public cloud. We focused on testing various functional units, including each side-channel primitive that would be incorporated into our final collection and attack code.

One critical component was the timer. For our purposes, it was essential that each chosen language had an accurate and high-precision timer available during runtime. The aim of most side-channel attacks is to discern the differences between cache miss and hit times across different cache layers and physical memory. Since caches closer to the CPU typically have much shorter response times compared to those further away, it was imperative to ensure that our timing primitives were precise to the nanosecond scale. We conducted multiple tests using the same dataset to determine a possible standard deviation, further verifying that these timers would function effectively across various parameters.

Understanding the cache memory size available on the server running our code was also crucial for conducting many side-channel attacks, as the easily readable cache size would likely be obscured by isolation barriers. We needed to monitor cache response times continuously as the system processed our code. Once we determined the amount of cache memory, we adjusted the code parameters based on the available cache. Similar testing procedures were employed to ensure our fetch times were not indicative of the cache levels above or below our benchmark. After estimating the cache size, we modified the parameters accordingly in all other primitives used.

3.2 INTERFACE TESTING

A number of interface points were tested during our project. The primary interface tested was the internal communication between components and primitives. This was essential to ensure that information determined by one primitive could properly influence the parameters of another primitive receiving this information.

We also tested the interface between the low-output data and the attack code, to verify that the data created and generated by both were accurate and functioned properly.

Finally, the interface back to the attack code with the updated parameters was critically assessed. We ensured that all components automatically adjusted based on the input received before attempting to leak data. This step was crucial to ensure that the leaked data was correctly captured by the waiting capture tool.

3.3 INTEGRATION TESTING

Maintaining well-commented and legible code was crucial in reducing integration challenges throughout our project. The project primarily focused on the execution of code, interpreting the results, and identifying specific lambda functions within a constrained serverless environment. Successful integration was defined by the attack code functioning correctly in this environment, specifically its ability to accurately determine cache hits and misses within an acceptable time deviation.

We adopted an Agile methodology, where testing and development occurred frequently, allowing for continuous improvements and adaptations. Given the predefined list of languages supported by serverless functions, each team member selected a language to explore and exploit potential side-channel vulnerabilities.

The ultimate measure of integration success was the attack code's ability to execute flawlessly in the serverless environment, delivering precise timing results and correctly identifying cache dynamics. We successfully identified certain lambda functions using our Python-based attack code, confirming the efficacy of our approach and the reliability of our testing processes.

3.4 SYSTEM TESTING

A critical component of our project involved ensuring that our code adhered to the operational constraints set by cloud providers. Although we used our own hardware for testing to enhance security, we built a system that closely emulated real cloud environments. Cloud providers frequently offer container images to facilitate local testing before deployment. We utilized these resources, along with AWS's Firecracker microVM software, which is designed to house the containers that execute the code.

In our tests, we were able to accurately emulate the cloud environment and conduct our evaluations without the need for actual cloud deployment. This setup allowed us to meticulously measure the time and space requirements imposed by the providers and verify that our code remained within these limits. Running the code locally also enabled us to segment and test each function individually. This granularity in testing was crucial for identifying specific sections of the code needing improvement or those that were not meeting the required constraints.

3.5 REGRESSION TESTING

To ensure that new additions did not disrupt existing functionality, we meticulously preserved functioning code both locally and on the server. This precaution guaranteed backups were available, even if code was accidentally deleted either locally or on the server. We also utilized GitLab for version control, allowing us to push all functional code to the repository, from where it could always be recovered if necessary.

A critical aspect of our testing was to ensure the uninterrupted operation of our server and the proper functionality of our Micro VMs and associated code. The server, which was shared with our professor and

not entirely under our control, was a vital component of our system design. Ensuring that the server operated correctly during all testing phases was paramount, especially in managing the language-specific coding tests.

One significant challenge we encountered was conducting tests when the server was not in use by others. Testing concurrently with other users often compromised our results, necessitating scheduling tests during off-peak hours to avoid interference.

Our design was driven by strict requirements. We needed to ensure that our code was written correctly and functioned as expected. Accurate data collection and the reliable performance of our server were imperative. Additionally, it was essential that our webserver accurately emulated Amazon's environment, reflecting the real-world conditions our system aimed to mimic.

3.6 ACCEPTANCE TESTING

Throughout our project, we consistently demonstrated that both functional and non-functional requirements were met by engaging frequently with our client, who also served as our primary point of contact. Our regular meetings were essential for discussing project goals and addressing any issues that arose during the development process.

We focused on collecting data that illustrated the specific security vulnerabilities associated with using Python in a serverless environment. This approach highlighted the effectiveness of our Python-based side channel attack methodology and its relevance to real-world security challenges.

As the school year progressed, we continued these demonstrations, showcasing how our Python code functioned and effectively collected data. This not only confirmed that our system performed as expected under simulated conditions but also provided opportunities to refine our approach based on feedback. These interactions ensured that the project's codebase was thoroughly tested and met all specified criteria set by our client.

3.7 SECURITY TESTING (IF APPLICABLE)

Security testing played a pivotal role in our project, ensuring that our web server, environment, and Python-based side channel attack code were secure and resistant to malicious use. Given the sensitive nature of identifying lambda functions, it was imperative to uphold high security standards to prevent any misuse of our methodologies or findings. We meticulously designed our webserver and testing environment to adhere to best practices in security. Throughout the testing phase, we utilized our attack code to identify some lambda functions.

3.8 RESULTS

Our team has successfully developed a side channel attack methodology to identify AWS Lambda functions based exclusively on their cache footprints. Our project aimed at understanding the cache impression of Lambda functions and utilizing this information to discern their identities.

Accomplishments: We engineered an attack framework, built a complete web server from scratch, and generated Lambda functions to test our hypothesis. This comprehensive setup allowed us to identify specific Lambda functions that exhibited distinct cache impacts. While not all functions could be detected—owing to their minimal cache interaction—those with significant impacts were consistently identifiable.

Technical Insights: The challenge lay in the diverse ways each Lambda function utilized CPU cores, affecting their cache footprints. Our approach involved meticulous analysis and replication of Amazon's serverless environment, which was crucial for obtaining realistic and actionable insights. The environment setup was particularly challenging but was ultimately successful, ensuring that our findings could hypothetically translate into real-world scenarios.

Results and Achievements: The project met its goal by identifying functions with pronounced cache impacts, demonstrating the feasibility of a cache-based side channel attack in an interpreted language environment.

Challenges Overcome: One significant hurdle was replicating an AWS-like environment to ensure our experiments mirrored real-world operations. Despite initial obstacles, our team adapted and overcame this by deploying customized solutions that mimicked those used by Amazon.

This project not only underscores the vulnerability of serverless functions to side channel attacks but also propels forward our understanding of cloud computing security.

4 Implementation Details

Experiment Setup

To conduct the side channel attack experiments, first establish a controlled environment that mimics a typical cloud server hosting serverless functions. This setup involved:

Server Configuration: A configured virtualized environment using hypervisors that simulate the cloud infrastructure. Multiple isolated virtual machines (VMs) were set up to represent different tenants running concurrent serverless functions.

Lambda Function Deployment: Deploying a variety of custom lambda functions written in Python. These functions are designed to vary in complexity and resource usage, from simple computational tasks to complex data processing operations, to represent real-world serverless applications.

Server Configuration

AWS Lambda functions run in a custom Amazon designed hypervisor called Firecracker which was built for speed and size allowing for MicroVMs with small footprints to run as part of the Linux Kernel's KVM subsystem. The server is hosting a Python Django web server that is responsible for process management, setup, teardown, and cleanup of these VMs. The setup process required to start a VM requires starting up a number of resources including creating a TUN/TAP network interface on the host machine for networking, bridging that connection to another interface for connection between VMs. This is all done using a virtual bridge network adapter that acts as a virtual ethernet switch between all the connected nodes. Specifically this is using a virtual bridge network adapter that is also a docker network to allow for additional AWS services such as Minio acting as S3 storage to be created as docker container and attached to it. This allows for the lambda functions to use these "AWS" services without needing to deal with multi-step routing which is resource intensive, and slow.

The next major step in the setup process is creating the actual base Linux file system that is used to boot the VM. This is done on the fly, as the Lambda code that is stored is placed into a new disk image that is then attached to the VM at the moment you press "start". In the event that the image has been built before it will use the pre-saved copy of the disk to save even more time booting, unless a rebuild is explicitly requested. This allows for rapid iteration and change to the Lambda functions as a simple press of a button just like pushing an update to the AWS cloud, while maintaining the flexibility of needing to store disk images for each function permanently. After the disk image is created, and ready to run, the MicroVM will be started automatically, and begin accepting requests.

At the end of VM running, the entire thing needs to happen in reverse, stopping the MicroVM, tearing down the networking, clearing the reserved space, and ending with cleaning up sockets and log files that Firecracker does not automatically clean up on its own.

The other major function of the backend is to act as a router between the real world and the lambda functions. As each MicroVM will take a randomly assigned IP address at boot, that means the request made to it each time will be changing. To prevent this, and to mimic the AWS environment, the server acts a router, that will take in requests, check if the specified Lambda function exists and is running when the request is made, forwarding it along to the proper IP address and endpoint if it is, then return the response that comes back. This also handles queuing of functions calls, so a second call to the same function instance will wait until the first has completed, with a timeout in the event the function is taking significant time to run, and prevents the closure and stopping of the MicroVM if there is still an active request being made, so that requesting client will always get a response from the lambda after the request is initiated.

Detection Techniques

The detection phase focused on identifying lambda functions being run.

Cache Behavior Analysis: By observing the cache access patterns and timings, one could infer the presence of other functions affecting the cache state. Significant deviations from baseline measurements indicated that their might have been some flaws in some of the returned results. For example, we would occasionally get a negative value.

Resource Utilization Patterns: Monitored CPU and memory usage patterns to detect anomalies that suggest interactions or interference between functions, which might also indicate shared hardware.

Exploitation Methods

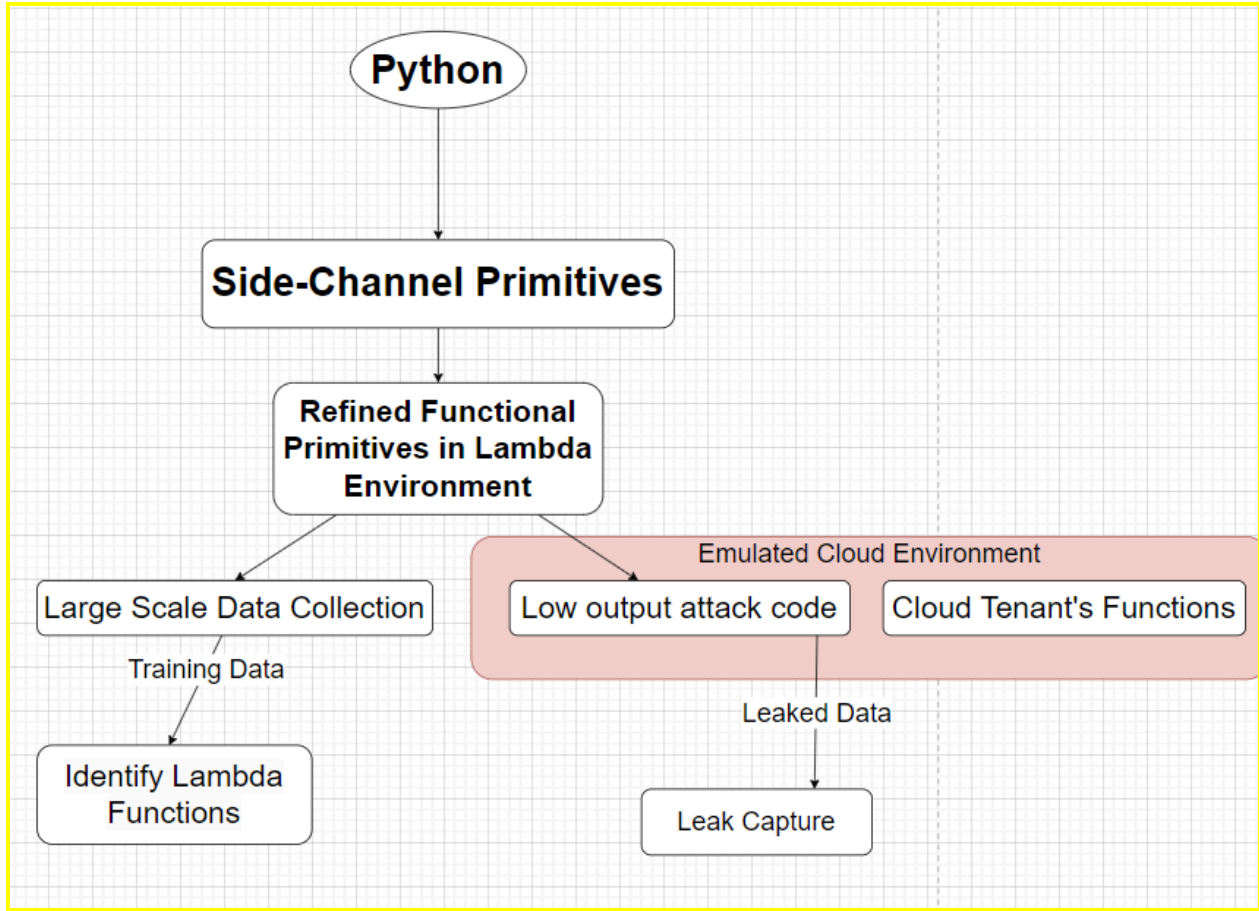
Experimented with several side channel attack vectors:

Time-based Attacks: Measuring the time it takes for certain operations to complete allowed us to infer about other processes running on the same server. For instance, increased execution time might indicate high CPU usage by another function.

Cache-based Attacks: Implemented strategies to observe and manipulate the CPU cache to extract information about other functions' activities. This involved creating controlled contention on the cache and measuring the impact on execution times of our lambda functions.

Conclusion

This comprehensive setup allowed us to not only demonstrate the feasibility of side channel attacks in a cloud-like environment but also test and verify the effectiveness of various countermeasures. The insights gained from these experiments are critical for improving the security frameworks of serverless computing platforms.



Functionality

Since our project was similar to a research project, the functionality was not entirely known. The data we collected could have been inaccurate which would have caused a lot of problems in creating our final report. However, we wrote code that worked properly, and used that code to collect accurate data about different lambda functions being run. We have more non-functional requirements since our data relies on efficiency, usability, and reliability. We are now able to explain the benefits and drawbacks of using lambda functions in a serverless environment.

5 Broader Context

5.1 Impact on Public Health, Safety, and Welfare

The research project on security vulnerabilities in serverless computing significantly impacts public health, safety, and welfare by enhancing data security and privacy for cloud service users. By identifying and mitigating risks like data leakage and insufficient isolation, the project protects sensitive personal and

organizational data, which is crucial for users storing critical information such as health records and financial data.

Preventing Malicious Exploits: Addressing vulnerabilities prevents potential malicious uses, protecting businesses and individuals from financial losses and service disruptions.

Regulatory Compliance and Trust: Improving security practices aids organizations in meeting stringent data protection regulations, enhancing trust in cloud technologies used in critical sectors like healthcare and banking.

Community and Educational Benefits: Secure serverless platforms foster economic and social benefits by encouraging innovation and providing a safer environment for new digital services. Additionally, the project has an educational component, improving developer understanding of security in software development, which indirectly benefits public welfare by promoting safer digital environments.

Overall, this project not only addresses technical security challenges but also enhances the digital safety and welfare of various stakeholder groups by ensuring more secure and reliable cloud computing environments.

5.2 Global, Cultural, and Social Impact

The project on security vulnerabilities in serverless computing systems extends its impact beyond technical realms, touching on global, cultural, and social dimensions. By enhancing security in cloud technologies, the project aligns with the values, practices, and aims of diverse cultural groups, reflecting a universal commitment to data privacy and cybersecurity.

Global Relevance: In a world increasingly reliant on digital services, the project's emphasis on robust security protocols in serverless computing caters to a global audience. With cloud services transcending national borders, improving these systems ensures that people and businesses worldwide can trust and rely on technology for essential services, fostering global digital resilience.

Cultural Sensitivity: The project considers the varied importance of data privacy across different cultures. In regions where data protection is highly valued and strictly regulated, such as the European Union, the project's focus on enhancing security and compliance aligns with local cultural and legal expectations, thus respecting and reinforcing these norms.

Professional and Workplace Impact: For professionals in the IT and cybersecurity fields, the project reflects the best practices and evolving standards of the profession. It provides a framework that other professionals can adopt, thus promoting a culture of continuous improvement and innovation within workplaces that rely on cloud technologies.

In summary, this project not only addresses technical security concerns but also considers the broader global, cultural, and social contexts. It reflects and respects the diverse values and practices of the communities it impacts, promoting a safer, more inclusive digital world.

5.3 Environmental Impact

The project on identifying vulnerabilities in serverless computing, though digital, has noteworthy environmental implications, particularly concerning energy consumption and electronic waste.

Reduced Energy Consumption: By optimizing server resource usage through serverless computing, this project supports the broader adoption of systems that require less energy for data processing and storage. Efficient server utilization helps reduce the overall energy demands of data centers.

Decrease in Electronic Waste: Enhancing the security of serverless architectures may extend the lifespan of server hardware, reducing the need for frequent upgrades and minimizing electronic waste. This contributes to lessening the tech industry's impact from rapid hardware turnover.

Indirect Environmental Benefits: Promoting serverless computing reduces the need for extensive on-premises servers, decreasing the energy used for power and cooling. This shift to cloud-based services can lead to a lower carbon footprint in IT operations.

In summary, while primarily focused on cybersecurity, this project contributes to environmental sustainability by promoting energy efficiency and reducing electronic waste through the enhanced security and adoption of serverless computing.

5.4 Economic Impact

The project focusing on enhancing the security of serverless computing platforms carries significant economic implications, both within the tech industry and broader economic realms.

Cost Efficiency for Companies: By improving the security frameworks of serverless computing, the project helps companies reduce the potential financial losses associated with data breaches and cyber-attacks. Secure serverless platforms allow businesses to trust cloud environments for their operations, minimizing the costs associated with managing and maintaining on-premises servers. This shift can lead to significant savings in IT budgets, reallocating resources to other critical areas of business development.

Consumer Costs: For consumers and small businesses, the adoption of serverless computing can translate into lower costs of digital services. As providers pass down the savings from reduced infrastructure and operational costs, consumers may benefit from more affordable, scalable, and secure cloud-based services.

Broader Economic Effects: On a larger scale, the project's impact on improving serverless computing security can enhance overall economic stability by safeguarding critical infrastructure and data. This protection is crucial for sectors like finance, healthcare, and government, where data integrity and security are paramount. Enhancing the security of these systems helps prevent potentially massive economic disruptions caused by cybersecurity incidents.

Reduced Economic Disparities: By making secure computing more accessible and affordable, serverless technology can help reduce economic disparities between large corporations and smaller entities. Small businesses and startups can leverage the same powerful tools as larger companies without the substantial upfront investment, leveling the playing field and fostering more inclusive economic growth.

In conclusion, the project's focus on securing serverless computing platforms has the potential to drive substantial economic benefits, ranging from direct cost savings for businesses to broader impacts on market stability and growth. These benefits underscore the importance of cybersecurity investments and their role in shaping the economic landscape of digital and cloud services.

6 CONCLUSION

This project has thoroughly examined the security vulnerabilities inherent in serverless computing systems, specifically those using interpreted languages. Through detailed analysis and testing, we successfully demonstrated the potential risks and data leakage scenarios that can arise when lambda functions are executed in shared server environments. The findings have emphasized the critical need for robust security measures to mitigate such risks.

Key Accomplishments: Developed an effective side channel attack that can identify specific lambda functions based on their unique cache footprints. This methodology not only highlighted the vulnerabilities in current serverless architectures but also showcased our ability to pinpoint areas for improvement in cloud security practices.

Challenges Overcome: One of the major challenges was creating an environment that accurately simulates real-world serverless computing conditions. Despite these hurdles, the team was able to replicate the cloud environment and perform meaningful security tests, which were essential for our research.

Future Directions: Looking forward, there is a clear path to refine and expand techniques. Further research is needed to cover a wider range of languages and execution environments. Additionally, future plans are to develop more advanced countermeasures against the types of vulnerabilities discovered, ensuring that serverless computing can continue to grow as a secure platform for modern software applications.

Impact of the Project: The outcomes of this project contribute significantly to the understanding of security in serverless computing, providing valuable insights for cloud providers, security experts, and software developers. The work not only enhances the security framework but also supports the broader adoption of serverless technology by ensuring it meets stringent security standards.

In conclusion, this project serves as a foundation for future efforts aimed at securing serverless architectures and promotes a deeper understanding of how interpreted languages can influence the security landscape of cloud computing. Through continuous innovation and rigorous testing, we can anticipate and counteract the evolving threats in serverless environments.

Review progress

As the project on security vulnerabilities in serverless computing environments reaches a significant milestone, it is crucial to reflect on the progress made, evaluate the outcomes, and consider the lessons learned. This review highlights the advancements, addresses challenges encountered, and measures achievements against the initial project objectives.

Achievements: The successful development and implementation of a side channel attack methodology capable of identifying lambda functions based on their cache impressions not only met the primary goal but also advanced understanding of potential security breaches within serverless architectures.

Methodology Refinement: Throughout the project, testing methods and diagnostic tools were refined to enhance accuracy and reliability. These adaptations resulted in more robust data collection and analysis processes, ensuring that findings are both valid and applicable to real-world scenarios.

Challenges Encountered: A major challenge was the complexity of simulating an accurate cloud environment for testing. Adapting the approach to overcome these difficulties was instrumental in maintaining the integrity and relevance of the research.

Lessons Learned: The project has been an invaluable learning experience, providing deep insights into both the technical aspects of serverless computing and the practical challenges of conducting security research in this field. It highlighted the effectiveness of collaborative teamwork, resource management, and navigating the complexities of innovative cybersecurity research.

Future Directions: With a solid foundation laid by the current phase of research, future work will focus on expanding the scope of testing to include additional programming languages and more complex serverless architectures. The aim is also to develop a machine learning model that can accurately predict what lambda function is being run based on the data it is fed.

Value of the Design toward the problem and users

The design developed in this project addresses critical security vulnerabilities within serverless computing platforms, providing substantial value both in solving prevalent issues and enhancing user experience. By focusing on the detection and mitigation of security risks, such as data leakage and cross-function interference, the design directly contributes to safer cloud environments. This improvement is crucial for users relying on these platforms for processing sensitive data and running critical applications.

Enhanced Security: The side channel attack methodology enables the identification of lambda functions based on their cache impressions, highlighting potential security breaches. This capability allows cloud service providers to better monitor and secure their environments, preventing unauthorized data access and ensuring the integrity of hosted applications.

Increased Trust: By demonstrating the ability to detect and address security vulnerabilities effectively, the design fosters greater trust among users. Businesses and developers are more likely to adopt and invest in serverless technologies knowing that they are protected against emerging security threats.

Scalability and Flexibility: The project's design supports scalability and flexibility, key attributes valued by users of cloud services. As serverless computing automatically adjusts resource allocation based on demand, the integrated security measures ensure that these adjustments do not compromise the system's security. This scalability ensures that the solution is effective for both small-scale applications and large enterprise systems.

Proactive Risk Management: The ability to foresee and mitigate potential vulnerabilities before they are exploited is a significant advantage for users. This proactive approach to security not only protects data but also helps maintain continuous service availability and performance, which are critical for business operations.

Potential Future Steps for Technical Development and Societal Value

The project's success in identifying security vulnerabilities in serverless computing paves the way for several future steps that promise to enhance technical development and provide value for users and society. These

steps include advancing machine learning models, testing in real-world environments, and broadening the range of interpreted languages utilized. Each of these initiatives aims to refine the effectiveness of security measures and expand the applicability and safety of serverless computing platforms.

Training and Testing a Machine Learning Model: A key future step involves developing a machine learning model to improve the detection of vulnerabilities. By training this model on vast datasets derived from various attack scenarios, it can learn to predict and identify potential security breaches more efficiently. Testing this model rigorously will ensure its accuracy and reliability in predicting threats, thereby enhancing the security protocols of serverless computing environments.

Testing Attack Code in Real Environments: To validate the effectiveness of the newly developed security measures and the robustness of the attack code, it is essential to conduct tests in real-world environments. This testing will help uncover any unforeseen issues and ensure that the attack code can operate effectively under various operational conditions. It will also allow for adjustments based on real user data and operational feedback, ensuring the solution is not only theoretically sound but also practically viable.

Expanding the Interpreted Languages Used: Broadening the scope of interpreted languages covered by the security analysis is another significant future step. By including more languages, such as Go, Pearl, or others commonly used in serverless architectures, the project can cater to a wider user base. This expansion will help address the unique vulnerabilities of each language, thereby providing a more comprehensive security solution that benefits a larger segment of the serverless community.

Providing value for Users and Society:

- **Enhanced Data Security:** By continuously improving security measures, the project ensures the protection of sensitive user data across various industries, including healthcare, finance, and government, thus supporting societal trust in digital services.
- **Economic Efficiency:** Improved security reduces the economic impacts of data breaches and enhances the operational efficiency of serverless computing, passing cost savings to users and boosting the economic viability of adopting cloud technologies.
- **Promoting Technological Innovation:** As the project expands to cover more languages and integrates advanced technologies like machine learning, it fosters innovation within the tech community. This not only drives technological advancements but also encourages a culture of security-first thinking in software development.

APPENDIX 1: OPERATION MANUAL

Initializing the requirements for the backend

The backend server requires the ability to do a number of privileged actions such as creating networking interfaces and mounting disks. There is an `inti.sh` bash script that will add a `sudoers` file that is needed to allow these privileged actions, download the required executables, download needed images, setup docker to allow for networking to the VMs, check the iptables rules that need created for the networking process, and create the access keys needed for the Minio server acting as an s3 bucket.

Because of the requirements needed with changing the `sudoers` files, and making privileged downloads, it is required to run this script as root, ideally with `sudo`, such as seen below.

```
$ cd src/backend
$ sudo ./init.sh
```

Creating the root filesystems

While the server itself has been initialized, there are no file systems that can be used to actually run lambda code yet; this is what this step will be doing. Inside “resources/build_image” there are a pair of scripts that can be used to create file systems for ruby and python lambda functions to run on the server. The scripts will take a base Linux image and transform it into a file system that automatically runs the lambda code at startup. To do this, it makes a number of changes to the boot sequence to allow for networking, sets up the startup process to spawn the lambda, mounts the external disk containing the lambda code, and copies the appropriate runtime files from a docker container made by AWS for testing lambda functions locally to make the environment as close as possible. After this copy, it will then make a number of minor changes to this code that allows for the automated passing in of the starting function instead of needing to be hard-coded like the docker image.

```
$ sudo ./resources/build_image/build_python39_image.sh
```

Creating your first lambda

The next major step that needs to be done to actually get your own lambda function running is creating the lambda itself. This will involve placing some files in the “src/lambda” folder that the backend will use to run the code on demand. The first step involves creating a new folder inside this directory for example “aws-python” which could contain a basic python lambda code seen below in a file named handler.py.


```
import json

def hello(event, context):
    body = {
        "message": "This is a sample message",
        "input": event,
    }




    return {"statusCode": 200, "body": json.dumps(body)}
```

This is the code that will execute each time your lambda is called with the event being the json data as a string that was sent to the function, and the context including information about how the function is running. Alongside this, there is another file that needs to be created for the server to recognize this as a lambda function; this file is “serverless.yml.”

```
# Required
provider:
  name: aws
  runtime: python3.9

functions:
  # The name of the function
  hello:
    # This is in the format <file name>.<function name>
    handler: handler.hello
```

This file will contain information about the function itself including the function name as the title item in the list, and the handler which has the format <file name>.<function name>. This is used to call the lambda when it starts. The provider section with “name: aws” is required, and the runtime is required to match the runtime you want to use for the function. At this point, once you start the server, that will be enough to cause the lambda to automatically show up on the main page.

 hello	4 CPU cores  python3.9 512 MB 1 CPU cores	 False Rebuild
---	---	--



Note how this example lambda was given 512MB of RAM and a single CPU core. Just as this can be changed when creating a lambda function, it can be changed here too in the serverless.yml file. By adding the cores or memory sections to the file underneath the specific function, it will create it with those constraints instead of the default.

```

# Required
provider:
  name: aws
  runtime: python3.9

functions:
  # The name of the function
  big-core:
    # This is in the format <file name>.<function name>
    handler: handler.hello
    # Set the number of CPU cores
    cores: 6
    # Set the amount of memory
    mem: 8192

```

 big-core	 python3.9 8192 MB 6 CPU cores
--	---

Using Minio S3 in a lambda function

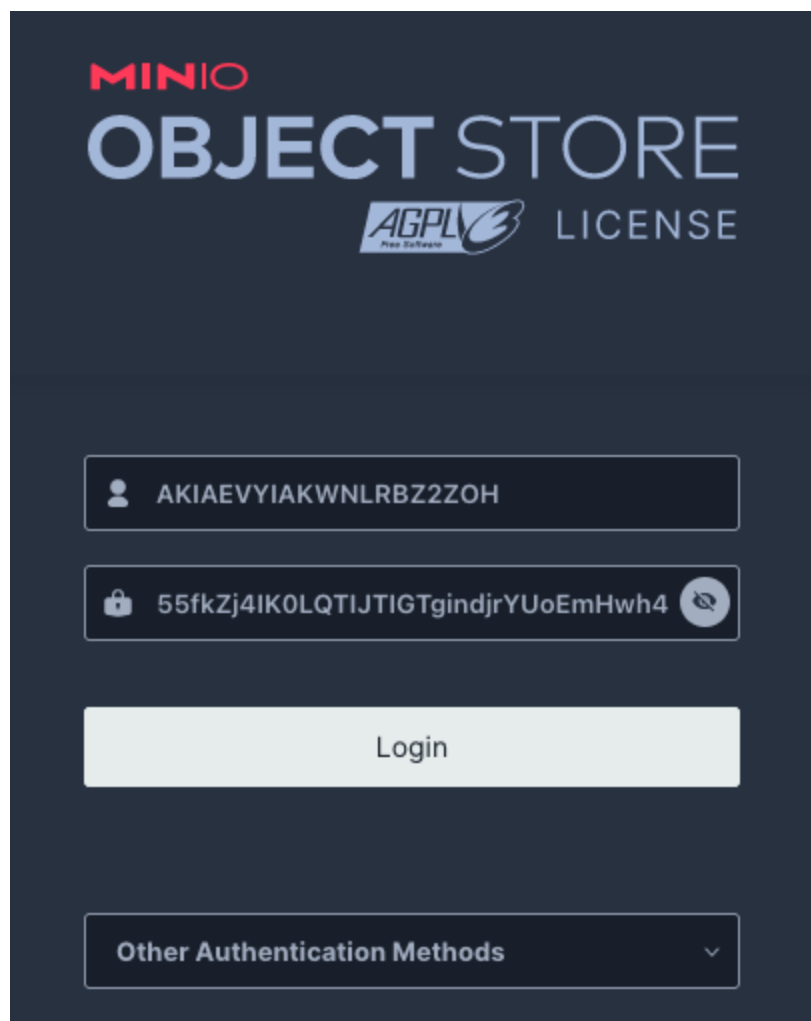
Part of the server running is that there is additional Minio server running, which is an S3 compatible storage system that allows for the lambda functions to transfer files. This does require a minor configuration change inside the lambda code you will want to run. The first thing that you will need to do is print out the secret key created as part of the initialization process. This is done by running the following commands:

```
cd src/backend
cat .env
```

This will print out two lines, one with “AWS_ACCESS_KEY_ID”, and one with “AWS_SECRET_ACCESS_KEY” these will be used in the code. The change the needs to be made in your lambda code is setting AWS’s boto3 client to use the custom endpoint of “<http://192.168.24.2:9000>”, and providing the custom access AWS_ACCESS_KEY_ID, and AWS_SECRET_ACCESS_KEY to the client allowing it to authenticate with the server.

```
import boto3
s3_client = boto3.client(
    's3',
    aws_access_key_id=minio_access_key,
    aws_secret_access_key=minio_secret_key,
    endpoint_url='http://192.168.24.2:9000',
)
```

At this point you can use the created `s3_client` as you would with a normal `s3` client from the `boto3` library. The same idea can be done for any code that is running locally and not in lambda, but replacing the endpoint url with “<http://localhost:9000>” while keeping the secret access key and the access key id the same. If you are on the server itself, you are also able to login to the Minio GUI by going to <http://localhost:9000/> in the browser.



This page will ask for login. The username to login with is the `AWS_ACCESS_KEY_ID`, and the password is the `AWS_SECRET_ACCESS_KEY`.

Running the server

At this point the server itself is ready to be run, and can be started. The simplest way to do this is with the script "run.sh" in the backend folder that makes it as simple as running `./run.sh`. However, if you want to start the server manually, you need these commands in order to properly setup the environment and start the Minio service.

```
#!/bin/bash

# Start minio
docker compose up -d

# Activate the virtual environment
source .venv/bin/activate

pip install -r requirements.txt

# Add the firecracker binary to the path
export PATH="`pwd`/data:$PATH"

# Run the server
python3 web-server/manage.py runserver 0.0.0.0:8000 --noreload
```

At this point, you can go to <http://localhost:8000> and see the following page with any of the functions that you created in the src/lambda folder below this. This is the main user interface for manually starting and stopping the functions themselves.

Currently Running VMs

VM Name	IP Address	Status	PID
Function Name	IP Address	Status	

These user interface options are rather simple and can be seen below. There is the button labeled “rebuild” to save startup time, the first time that a function is run a new filesystem to store that function is created and saved in the temporary directory. However, that means if there is a change in the files since the last time that it was started up, then it will not reflect those changes. This button will cause that filesystem to be rebuilt, so that it can reflect any changes that were made. The second button start is a way to manually start the VM, the firecracker VM that contains the lambda function.

 hello	 python3.9 512 MB 1 CPU cores	 False <input type="button" value="Rebuild"/>	<input type="button" value="Start"/>
---	--	---	--------------------------------------

Running a Lambda Function



After starting a function, you can make a traditional function call straight directly to the server on the endpoint http://server-name:8000/function/<function_name>/. Do note that the ending slash is a requirement, and will result in an error if that is not given.

```
snmay24_16@berk:~$ curl http://berk.ece.iastate.edu:8000/function/hello/ -d '{"test": "string"}'
{"statusCode": 200, "body": "{\"message\": \"Go Serverless v3.0! Your function executed successfully!\", \"input\": \"{\n\n\"test\\\": \\\"string\\\"}\"}"}snmay24_16@berk:~$
```

Here is an example of running the hello function from earlier by using curl to make a request to the server, and it responds back with the value that we passed in.

Debugging a Lambda Function

At some point there might be a weird error that happens like your lambda function not returning any data. Since the VM that function runs is fully networked, it is possible to ssh into this virtual machine and check out the logging of the VM to understand what is going wrong. The first step is getting the IP address of the machine which can be seen on the main website page.

Function Name	IP Address
 hello	 192.168.24.12

After this the ssh key that needs to be used to login is the data section of the webserver. So you can cd in the “src/backend” folder again. Then run the command below to actually ssh into the VM.

```
ssh -i data/web-server/keys/ubuntu-22.04.id_rsa -o StrictHostKeyChecking=no root@192.168.24.12
```

From here it is possible to use many of the traditional linux commands to go through the debugging process. The most important detail to know though is that there is systemd service that causes the lambda to start at boot which is named “lambda”, so you can check the current status of that service with “systemctl status lambda”, or check the log history of it with “journalctl -xeu lambda”

Using the API to start and stop functions

The starting and stopping of the functions does not have to be done manually however, and there is an easy API that can be used to start and stop them instead. It is simply making a POST request to the start, or stop end points respectively where the parameter is the name of the function to be started or stopped as seen in the 2 curl commands below.


```
# Start the function
curl -X POST http://berk.ece.iastate.edu:8000/start/ -F function="hello"
# Stop the function
curl -X POST http://berk.ece.iastate.edu:8000/stop/ -F function="hello"
```

How to run multiple functions side-by-side

The server fully supports running and routing multiple functions at the same time. This can best be seen in the short script below how to do that. First, you will start up both of the functions using the start endpoint, then make a request to each of the functions, with any function that you do not want to wait on being backgrounded with the “&”. This code will wait for function2 to complete, then attempt to stop the VMs. This stop is a blocking request that will wait for the Lambda to have no open requests to it before actually stopping the VM that is running that lambda. This will result in both of these functions closing only after they have both completed.

```
# Start the pair of functions
curl -X POST http://berk.ece.iastate.edu:8000/start/ -F function="function1"
curl -X POST http://berk.ece.iastate.edu:8000/start/ -F function="function2"

# Make a request to function1, but background it, then start a request to function 2
curl http://berk.ece.iastate.edu:8000/function/function1/ -d '{"some": "data"}' &
curl http://berk.ece.iastate.edu:8000/function/function2/ -d '{"some": "data"}'

# Stop the functions, both of these are blocking calls that will not stop til not requests exist
curl -X POST http://berk.ece.iastate.edu:8000/stop/ -F function="function1"
curl -X POST http://berk.ece.iastate.edu:8000/stop/ -F function="funciton2"
```

How to clean up in the event of an error

While the server does its best to always clean up after itself by closing sockets, stopping processes, and releasing handles. There are times where if there is a sudden stopping, such a disconnect from the person running the server, that it will leave certain resources partially open. It is at the point where none of the resources that are left open will use significant resources on the system, but it will prevent the server from running properly the next time that it is run. For this, there was a script that was made, and is in the “src/backend” folder named “clean.sh” that will go through the cleanup process cleaning up any of those open resources that got left behind. If you happen to see an error such as a “socket error” or an “already in use error” or some other odd error that appears to have no real cause, this is the best first option to begin troubleshooting the issue.

APPENDIX 2:

APPENDIX 3:

APPENDIX 4:

Gitlab Repo: <https://git.ece.iastate.edu/sd/sdmay24-26>