Information Flow Control for Serverless Systems

Rishabh Chawla
The Pennsylvania State University
May 2020

***Abstract*—Security for Serverless Systems is looked at from two perspectives, the server-level security managed by the infrastructure company and the Application level Security managed by the tenants.The Trusted computing base for cloud systems is enormous as it encompasses all the functions running on a system. Authentication for systems is mostly done using ACL. Most Serverless Systems share data and thus, ACL isn't sufficient. IFC using appropriate label design can enforce continuously throughout the application. IFC can be used to increase confidence between functions with other functions and cloud provider and also mitigate security vulnerabilities making the system safer. A survey of the present IFC implementations for Serverless Systems is presented and system designs which are relevant to Serverless Systems and could be added to Serverless Systems Architecture and, an idea of an IFC model that could be effectively applied in a decentralised model like serverless systems.**

***Keywords*—Information flow control; serverless systems; language based security; cloud computing**

I. Introduction

Serverless Systems are systems where functions owned by tenants are executed when the functions are triggered, on platforms managed by the cloud provider. The infrastructure, security and updates of these systems along with the hardware are managed by the cloud provider and the tenant only manages their function and it's security. Security is looked at from two perspectives, the server-level security managed by the infrastructure company and the Application level Security managed by the tenants. The reasons for the boom of serverless computing are elastic scalability, ease of deployment, and flexible pay-per-use pricing. Trusted computing base(TCB) consists of all the parts of the system (like hardware, software, libraries, firmware), all the components which could leave the system vulnerable and jeopardize the security of the whole system. The TCB for cloud systems is enormous.

A. Server-Level Security

Serverless Systems are cost-effective resource sharing platforms where the tenants only pay for the time their function/service is executing/working on the machines and thus, the machine setup time for a function i.e the microVM/container creation and startup time has to be minimal, as that time is paid for by the cloud provider, which does not leave a lot of scope for setting up security measures specific for functions by the cloud provider. The cloud provider's platform manages function placement and scheduling, automatically spawning new function instances on demand. This also means that multiple functions are run on the same server, the functions owned by different teams/companies with no security guarantees to each other, which leaves the possibility for side-channel attacks, and attacks specific to those applications/services which might leave the host machine vulnerable. Traditional security practices are unable to achieve the flexibility, generality and efficiency expected by cloud providers and tenants [1].

B. Application Level Security

Users express their applications as collections of functions triggered in response to user requests or calls by other functions. With serverless systems the use of third party services has increased which in turn increases the risk of data vulnerability during communication, the security of the third party services, the storage of keys used for communication with the services. [2] Applications need to consider security from the perspective that they are vulnerable to exploits of the third party apps, infrastructure, other tenants sharing the system, among others. Serverless Systems removes the burden of managing Server Level Security for the Application Development Teams as most tenants believe the cloud provider they are using, though measures could be taken to increase this confidence.

C. Paper Outline

Section 2 describes the Background on some of the different parts used in serverless architecture and basic idea of attacks specific to serverless systems. Sections 3,4,5 are the motivation to use IFC on serverless systems and explain the need for IFC and the advantages it could bring. Section 6 explains some IFC ideas which could be applied to different parts of the serverless architecture. Section 7 describes some serverless, cloud and general IFC implementations. The cloud and general implementations are on system parts which are part of the serverless architecture and could be modified to work on serverless architecture. Sections 8 and 9 explain the advantages and remaining questions after adding DIFC to serverless architecture. Section 10 is my idea of how all the mentioned ideas and implementations could be implemented together to setup a serverless DIFC system. Section 11 explains some Future Research ideas/direction.

II. Background

Many functions are run on a single bare-metal machine in Serverless systems, to improve security a virtual machine or container is used to execute the function, so two functions are basically running on virtualized environments rather than on the bare-metal machine itself and thus, separated by an extra layer of abstraction and thus, increasing security and making it harder for the functions to affect or read each other's information. We explain some exploits and mitigation techniques used in serverless systems.

A. Containers

Functions are hosted inside containers as containers encapsulate all the underlying software required for the application to run and are useful when applications need to run on different environments/machines. Containers fall short as they use the host operating system kernel, which means that there is a fundamental trade-off between security and code compatibility as, Container implementors can choose to improve security by limiting syscalls, at the cost of breaking codes which require the restricted calls [3].

B. MicroVM

MicroVM are minimilistic virtual machines. The idea behind microVM's were to protect against privilege escalation, information disclosure, covert channels and others. With microVMs which take less than a second to boot up, the security measures are enhanced. Adding advanced security features affects the performance of the system and might not be implemented by tenants in lesser security demanding environments [3]. To give an idea, Firecracker checks that the host kernel has mitigation enabled for Kernel Page-Table Isolation, Indirect Branch Prediction Barriers, Indirect Branch Restricted Speculation and cache flush mitigation against L1 Terminal Fault among others. [3]

C. Scheduling or Warm Containers

Each function invocation should ideally take place in a fresh environment, such as a container that is immediately destroyed after it's execution but to reduce the cost of setting up an entire runtime environment for each function execution, warm containers are cached and reused for future invocations of the same function within a pre-configured timeout window [4]. Opaque platform policies and scheduling algorithm details obscure this practice, making it difficult for customers to account for such issues during application development. Attackers can get their function on the same machine if enough functions are deployed [5].

1) Device Drivers: VM's use paravirtualised device drivers which interact directly with the VM host via an agreed channel. The alternative to this is a way slower virtual hardware using the native device drivers. Cloudburst describes the vulnerability in VM display functions of VMware Workstation that could be exploited by a video file to take over the operating system [6].

D. Hypervisors

Hypervisors are used to create virtual machines on bare-metal machines. KVM is a virtualization module for linux and is a type 1 hypervisor. It is used in Firecracker. Virtunoid is a privilege escalation exploit on KVM made in 2011 because of a missing check on the KVM emulation of PCI device hotplugging, which is used for devices which don't support being unplugged but when unplugged left a corrupt state and dangling pointers [7]. These kind of vulnerabilities are being mitigated by Hypervisor verification [8].

E. Attacks on Serverless Systems

Serverless systems contain functions which generally last seconds, thus, it is harder to attack them but there are exploits made for this kind of system. Rapidly ex-filtrate stolen data [9], cross-tenant side-channels [5] are some attacks for this type of system. Persistent function compromise is possible by malware in an in-memory partition of the system, is another example of an attack for this system. Attackers can also take advantage of the cloud providers warm container reuse policy to cache a compromised copy of the function that persists across invocations [9]. Logging and debugging support in serverless platforms lacks the ability to monitor a serverless application as a whole and therefore struggles to trace sophisticated attacks, for example an attack that depends on two executions of the function. [10]

III. Present Security Solutions for Serverless Systems

Present security solutions include language run-time libraries which are used to secure a single function according to developer defined policies as part of the source code. Static analysis of function source code could be used to detect violations of the principle of least privilege [11] and checking function dependencies against vulnerability databases [12]. Function developers rarely consider and secure interactions between functions, giving rise to emergent attack vectors such as API-based data exfiltration. There are products that model function behavior using machine learning to detect anomalous behaviors or wrap function event handler wrappers to inspect specific activities [13]. There are also run-time protections which include machine learning based detection of anomalous function behaviors [13] to prevent event-data injection prevention by inspecting incoming function invocation requests using existing penetration testing techniques like sqlmap. Runtime semi-automated troubleshooting based on log data, to ease reasoning about function behavior is present to make auditing easier [14].

IV. Lack in Present Security Solutions for Serverless Systems

A lot of pre-compiled third-party objects and proprietary closed-source functions don't provide source code access which is required by many of the present security techniques. The present security solutions are largely function-centric and their efficacy depends on the correctness of policies written by the function developers, complete access to source code and configuration files, and the compatibility of the tool with the functions specific language runtime, platforms, and event sources. Existing monitoring techniques offer limited observability into the interactions between functions and most of these monitoring services are limited to strict specified/available conditions. [13] Static check tools aren't able to detect implicit flows. Cross invocation attacks [13] aren't considered by present security solutions which occur between containers and also among reuse of containers (warm starts). A major and serverless specific problem is lack of proper function isolation [10]. Event injection attacks may target the function source code which might also leak other secrets stored in the container [15]. Azure Functions had an exploitable placement vulnerability, which led to the exploit to run arbitrary binary

code in containers making them vulnerable to many kinds of side-channel attacks [5].

V. Need for Discretionary Access Control

Major authentication check in today's world is done using Access Control List(ACL), also called Role-Based Access Control(RBAC). ACL has some limitations and vulnerabilities which can be fixed using IFC. It may be possible to bypass ACL checks, especially in web-based systems [16]. ACL does not implement any further control once the data has been authorized at entry point or discrete check point by checking the users allowed permissions. The application is trusted not to leak the data after the check. As there is a lot of data sharing among applications there is a need for controlling data flows between applications which may also send data ahead and these checks can be done using IFC [17]. Data can propagate or influence system behaviour indirectly in ways that aren't disclosed, which access control barriers at discrete points in code do not detect, while IFC using appropriate label design can enforce continuously throughout the application [18] IFC [19] could be used to add security policies to data and use these policies at run-time to control where user data flows. Since IFC security is linked to the data that it protects, both tenants and cloud providers can agree on the security policy, in a manner that does not require them to depend and rely on the particulars of the cloud software stack or application stack in order to effect enforcement [1]. IFC [20] provides a means to control and monitor data flow continuously, according to policy which could restrict that the data be restricted to a certain location in favour of laws [1]. IFC mechanisms can help enforce non-interference policies mitigating the fact that another system running on the same machine may observe the public outputs. IFC supports isolation of individual users data, and inter-tenant isolation [1]. IFC protects information by a global security policy that cannot be overridden by a misconfigured application. The policy explicitly and concisely captures constraints on end-to-end information flow through the system, majorly protecting the system by system calls and restricting the data flowing outside the network. The IFC system enforces the policy even for buggy or malicious applications, thus removing application code and configuration from the TCB of the cloud [21]. This case is valid when the right policy check parameters are set inside the application as used in the cloud infrastructure. A generic model to detect type could be made which could make this model stronger. Specifying an effective security policy is a difficult problem, failure to adequately restrict flows violates the principle of least privilege and leaves the system vulnerable but defining overly-restrictive rules prevents the correct operation of the system, thus increasing the development and testing time and requires checking all expected flows [22].

VI. IFC System Design with Serverless

This section contains general ideas which people have mentioned in regards to Cloud Computing/Serverless Systems and given a general idea of how IFC could be useful in solving them.

A. Warm Starts

Container creation accounts for a major chunk of time in the response time for a function after it was called/triggered and cost for container creation time is not paid by the tenant and is covered by the cloud provider and thus, cloud providers use warm starts which is reusing the container which was recently used to run the function, so that on another function call of that same function in a certain time limit, the container is reused rather than creating another container, so that the response time for the function is reduced and the cost for the container creation doesn't need to be paid. The cloud provider kills a function after a certain time limit as the tenant only pays for the time when the function is being used and not when it is idle and waiting for a request. It is expected that a serverless function activation handles a single request on behalf of a specific user and only accesses secrets related to this request. Each invocation starts from a clean state and does not get contaminated with sensitive data from previous invocations. Any state shared across invocations must be kept in a global data store. Warm startup is done using the method that after the initial invocation is complete, but before the actual function process starts, the process is forked and the function is run on a child process of the same process and purged after it's completed and this process is repeated again when the function is called again and thus, another child process runs it. This way the address space is in the child space and will not affect other processes that are run on or from it [21]. This way could be vulnerable when two child processes are running at the same time, as there isn't a lot of separation in that case, but in serverless system this isn't done. The child process would have to strictly be limited to it's address space as a this could be used for cross invocation attacks. Another way is tainting the sensitive data which is used for file access, and deleting all the tainted data after the function execution ends before the next function is executed on the same container [13]. We could also taint all the changes made to the filesystem during the function execution and revert them using something like a git svn or snapshot but that would have a higher overhead, so one could use tracking on all the changes on the filesystem which uses more processing power and could increase the execution time of the function, the time could be reduced by using even more processing power. One of these two methods could be used based on the trade-off of the time it takes after the execution of the function compared to the other one taking extra processing during function execution.

B. Termination

A container is created everytime a function is called/triggered and thus, for serverless systems the termination of a function can be as many times the function is called which is generally a lot. IFC Systems which leak information through the termination channel, where one bit of information can be observed by observing the termination or non-termination of the program. The parallel nature of the serverless environment amplifies this weakness, allowing the attacker to construct a high-bandwidth information channel, effectively defeating the purpose of IFC [21]. The termination channel present in most existing IFC systems can be arbitrarily amplified via multiple concurrent requests, requiring a stronger termination-sensitive non-interference guarantee, which can be achieved using a combination of static labeling of serverless processes and dynamic faceted labeling of persistent data [21]. We can use the security property termination-sensitive non-interference (TNSI) to

eliminate this channel [23]. SLam Calculus is used in the TNSI security property to achieve termination insensitive IFC model [24]. A way of achieving this in serverless systems is a combination of static program labeling with dynamic labeling of the data store, based on a faceted store semantics. Static program labeling restricts the sensitivity of data a serverless function can observe ahead of time, and is used to eliminate the termination channel. Dynamic data labeling is important to secure unmodified applications that do not statically partition the data store into security compartments, while the faceted store semantics eliminates implicit storage channels also [21].

C. Sticky Policies

In serverless systems functions are short lasting and can be run on any machine at any time, and a lot of them could be running at the same time triggered by different users. Data could be required for these functions to run which they get from databases or filestores (S3 and dynamo for AWS). At a higher level, sticky policies could be used to achieve end-to-end control over data. In sticky policy systems, data is encrypted along with the policy on that data. To obtain the decryption key from a Trusted Authority (TA)(In this case the database of filestore), a process must agree to enforce the policy. This agreement may be considered part of forming a contractual link between the data owner and the process decrypting the data [25]. A logging system of the decryption could represent a starting point of data flows and can be used for tracking.

D. Continuous Checks

Serverless systems applications are made using a combination of functions where the data flows from one function to another, and may also flow to third party functions for various reasons like verification and so on. In a system like this where data flow continues and the limitations of the data flow inside the intranet or outside isn't known, Continuous data checks could be used to check that data is only used at authorized places. This is done by storing the data with its label and any time the data needs to be accessed its label is checked with the process label and only if permitted, read/write operations on the data are permitted. This system could be implemented by the function sending the data, by checking that the function receiving has enough access, the storage location would also check the same. The function receiving could use the label of the function they got the data from for confidence label on that data. Thus, having security perspective by the one sending and also, the one receiving the data. A function call is given the label of the caller and actions allowed to the caller are only permitted to that execution. Similarly any data store being modified also stores the label with which it was modified and stores allowed to a particular label are read or written by that label. If any operations need to be done outside of the permitted value of the caller, declassification [26] is used.

E. Implicit Storage Channel

A serverless function always runs on behalf of a specific user and can be assigned a corresponding security label. The function's label determines its view of the data which it reads or writes in databases or filestores, the function can only observe the existence of data whose label does not exceed the function's label. In a situation where multiple functions with incomparable labels write to the same store location(database or filestore). We avoid information leaks in this situation by employing faceted store semantics, where each record can contain several values with different security labels [27]. Implicit storage channels is when the attacker infers secrets by observing the labeled values exist within particular store locations without observing the actual values [28]. An attacker could check that a location contains sensitive data by writing to a particular location and reading from there. One could block writing to that location but that would also leak that it contains sensitive data. One alternative would be to have extra data copies for different labels. This is shown by data store semantics where each record can contain multiple values. Though this has a high runtime cost [29].

F. Audit Logs

If an IFC system is made at the cloud level, including the network, OS and continuing to the application/function level, all this data flow can be used as a logging system. Enforcement of IFC can provide the opportunity for recording flow decisions to build a provenance like audit graph. This can be analysed to understand where, how, why and by whom the data was manipulated within the system. This audit data, captured during IFC enforcement, can help to demonstrate compliance with regulations by providing tangible traces, showing how the data was handled [30]. Under a conservative assumption that all secrets obtained during function execution propagate to all its outputs, we can track the global flow of information in the system by monitoring inputs and outputs of all functions in the system [21]. Audit Logs are made by tracking all information flow using tainting. An important detail is to get all the information before logging which would be necessary for analysing, thus, the point where the logs are stored needs to be as late as possible to get the most logs and we also need to consider the high performance penalty cost for it and minimize it. Major challenges with this system is being able to track all information flows and the logs being enough to recreate the situation or analyse the situation completely [31].

VII. IFC Implementations for Serverless Systems

A. Hardware Level - General Implementation

RIFLE [32] translates normal binary code to run on hardware with IFC tracking. Dynamic information flow tracking can be used at this level for checking the use of spurious values being used as instructions or pointers [33]. In serverless systems this method any attacks on the underlying hardware where the calls are sent to using the virtual machine can be checked.

B. Kernel Level - Cloud Implementation

Information flows in a system are only generated through system calls and shared memory between processes. If shared memory is restricted then information flows could only be generated using system calls. The entities defined in this model are processes, files, pipes and sockets. Privileges are only associated with processes(active entities). All labelled entities are allocated their labels when they are created. For a process

creating some entity the sub-rules associated with the flow are that the created entity inherits the labels. Certain processes have privileges, allowing them to change their labels that is, those processes are able to change their security context using declassification. The labels of passive entities (files, pipes and sockets) can't be changed. Processes are further associated with privileges over their tags. System calls creating flows are intercepted and IFC constraints are applied, enforcing IFC according to the labelling, other system calls are left unintercepted. The cloud tenant decides the labels and tags for processes and calls [34]. This case is valid in serverless systems as we run only one function inside a microVM and if there is any information flow between functions or third party services it has to go outside the function/microVM using system calls, as nothing else is running on the microVM thus, no shared memory among usage. This system can be used to restrict any malicious code injected from sending data outside the microVM and thus, restricting the outflow. The access to the microVM is protected and restricted by the cloud provider.

C. VM Level - General Implementation

Argos [35] modifies the QEMU(type-2 hypervisor which can be combined with KVM to make a type-1 hypervisor) virtualisation framework to extend the target code so that it defines isolation regions and checks information flow meta-data. It uses dynamic taint analysis to detect exploits and protects unmodified operating system processes. It checks the network data throughout execution to identify their invalid use as jump targets, function addresses, instructions. It has a very high overhead but could be used to find signatures which can be used with almost no overhead to find exploits during run-time [35]. This system is used to protect the VM itself so that the VM runs and terminates smoothly. This system could be used to confirm that the VM is restricted to the allowed and approved capabilities. Running this system on actual machines doesn't seem feasible because of the overhead but a system like this is very useful to actually analyse the new attacks that are being created and get their signatures to block them on actual running machines.

D. OS Level - General Implementation

If IFC is enforced at OS level, the applications running above the OS, run under the policy constraints expressed by the IFC labels tags. They do not need to be trusted not to leak data through the monitored labels in the processes [36]. This system has DIFC implemented at the granularity of processes, and integrates DIFC controls with standard communication abstractions such as pipes, sockets, and file descriptors via a application level reference monitor. This interface helps programmers secure existing applications. This system enforces the DIFC policy during runtime. The application consists of two types of processes. Untrusted processes are generally used for most of the work. They are constrained by, and maybe unaware of the DIFC controls. Trusted processes are aware of DIFC and setup the privacy and integrity controls that constrain untrusted processes. Trusted processes also have the privilege to selectively violate information flow control for example, by declassifying private data, or by endorsing data as high integrity. The system represents each resource a process uses to communicate as an endpoint, including pipes, sockets, files, and network connections. A process can specify what subset of its privileges should be exercised when communicating through each endpoint. Uncontrolled channels are modeled as endpoints that exit the DIFC system. This can be used as a security policy for end-to-end integrity protection. It can pull third-party plugins into its address space, but with end-to-end integrity protection, users can enforce that selected plugins never interact and potentially corrupt sensitive data, either on input or output [36]. This system can also be used by the cloud provider to restrict the usage of the functions as the microVM, the OS and underlying hardware is managed by the cloud provider. The cloud provider can have different levels setup and the functions can decide the amount of freedom they require and the cloud provider could isolate categories of functions in their own DMZ (Demilitarized Zone).

E. Network Level - Serverless Implementation

This system has agents residing in function containers to monitor storage and network behaviors. These agents dynamically generate taint labels that describe each function's file accesses and network requests. These labels are reported to a centralized controller. The controller then aggregates this information to discover the flow paths of the application. The information flow monitored by the controller can be restricted based on the security policy. Function calls are monitored with taint labels that and called/triggered using REST based APIs in this system. The system works with deploying a transparent forward proxy in each container that begins proxying network requests when the container starts. The network proxy performs network level tainting. The proxy inspects the REST call to determine appropriate labels with which to taint the current flow, which are mostly mentioned in the Rest call itself, and are compared with the policy file. Each invocation request is a unit of work in FaaS, functions are short-lived and taint labels are assigned per request. The "taint explosion" problem occurs because of this. The network taints can be summarized when a function makes multiple calls to the same domain, thus compressing the taint labels. Function level operations are checked across workflows to capture inter-function security violations. There can be Data leak through the network as Static network policies are bypassed by passing data to downstream functions with network access. This can be mitigated with Network level taint tracking. Another type of attack could be the Cross invocation side channel where residual data in warm containers is leaked across invocations and this can be mitigated using File access taint tracking and function garbage collection [13]. A similar design is mentioned in [37] where labels are used on data and communication between libraries is done using messages and based on the level of the process any data in the message above that label level is removed. Continuing on the previous system, the agent contains a system call tracing mechanism for monitoring function file I/O, allowing the system to detect cross-invocation flows resulting from container reuse. After the function finishes execution, all data on disk that was modified by the function is erased from the container. As current attacks require an explicit data flow from one function execution to another this procedure is sufficient to deny cross-invocation capabilities to the attacker. Commercial platforms provide only a small writable partition using an in-memory filesystem, the approach is significantly more efficient than

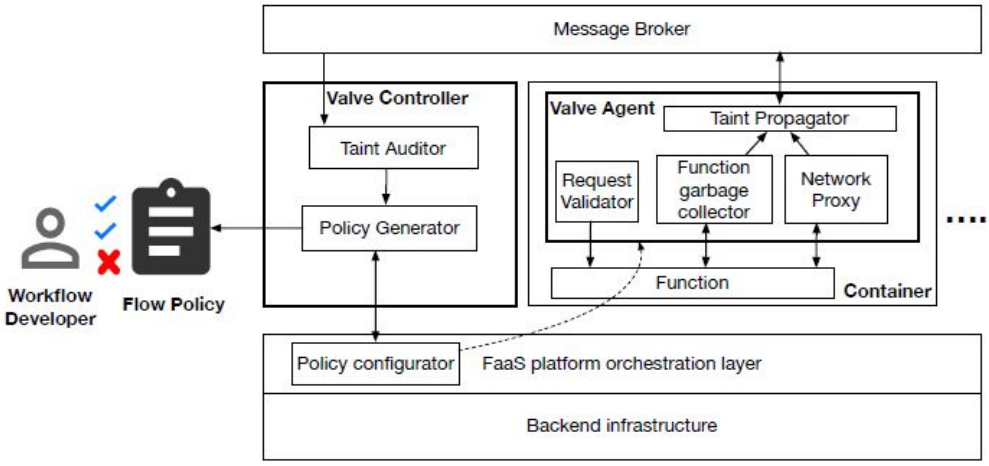

Fig. 1. Network Layer Design Architecture Presented in [13].

destroying and re-provisioning the entire container. The agents also check taint labels on file accesses and file access behaviors as they may violate security constraints of the application. Garbage collection is performed after each function invocation, the set of modified files are not tainted by previous function executions [13] (Fig. 1).

F. Service Level - Cloud Implementation

The system's Model generator component is responsible for building a model that semantically simulates the lifecycle or runtime execution of the candidate application. Then, the IFC engine performs information flow analysis on the unmodified application's bytecode with the help of the generated model. The vulnerability detector pinpoints insecure flow paths that violate data integrity and confidentiality. The result publisher component refines and reports the analysis results to the cloud provider and tenant. Based on the results, it decides if a security certificate should be granted to the candidate application, which is sent to both the cloud provider and tenant [38] (Fig. 2). The system uses IFC based on System Dependence Graph (SDG) and program slicing techniques for security inspection. The SDG has the advantage to model the information flow through a program by capturing both data and control dependencies [39]. Even with this model, the system is prone to stealth type attacks. This system can be added to serverless systems as a separate utility which could be used when any abnormality is detected to check the request, because this system has a very high overhead.

G. Application Level Implementation on Hadoop using Inlined Reference Monitor (IRM) - Cloud Implementation

IRM [40] implementation carefully leverages object encapsulation, control-flow safety, and type-safety properties of the binary language in which the function code is expressed, to guarantee that the surrounding untrusted code into which the IRM is in-lined cannot corrupt or circumvent the IRM's security programming at runtime. IRM whose programming is in-lined into untrusted binary jobs as they arrive at the cloud edge. After in-lining, the modified jobs self-enforce the security policy. The in-lined enforcement code maintains and consults an information flow graph (IFG) implemented as a distributed data resource within the cloud. The IFG tracks information flows between the various principals, and the IRM prohibits job operations that introduce explicit flows that violate any defined policy. This makes it easy to implement and adapt to real world clouds, since the cloud and the enforcement can be maintained completely orthogonally. It achieves this by enforcing an IRM that is in-lined into untrusted binary jobs at the cloud's edge. The resulting jobs self-monitor their accesses and collectively maintain a distributed information flow graph within the cloud, which tracks the history of flows and prohibits policy-violating operations. Well-established IRM design methodology is applied to secure the IRM against attacks from the code into which it is in-lined, protecting it even from threats that know all the IRM's implementation details. This system is limited by enforcement of mandatory access controls of explicit information flows between principals [41]. A system like this is hard to implement in serverless systems as the functions last under a second and aren't running all the time, so IRM would have a heavy startup overhead and wouldn't be very useful as it isn't running all the time, else it'll have to converted to a system which stores its state in a database and retrieves it everytime it starts up. I think that the cost and performance overhead would outweigh it's benefits. A system like this could be implemented at the cloud infrastructure level where the cloud enforces some principals and based on environments which need more security could have more/stricter enforcement which would increase the processing and runtime of the function and thus, the cost. This system will be useful at a cloud level for applications deemed dangerous by the cloud and yet requiring a lot of privileges. This system could be used to limit these applications from a moral and legal perspective. The idea of this system is to enforce a system that does not believe the cloud infrastructure. We could consider an implementation of a web-based application, where the front page is always running on a low system with an IRM system enabled and every other process is done using serverless systems, and thus, the scaling and running of all the serverless systems will be managed by the cloud infrastructure itself and these functions, would connect and interact with the

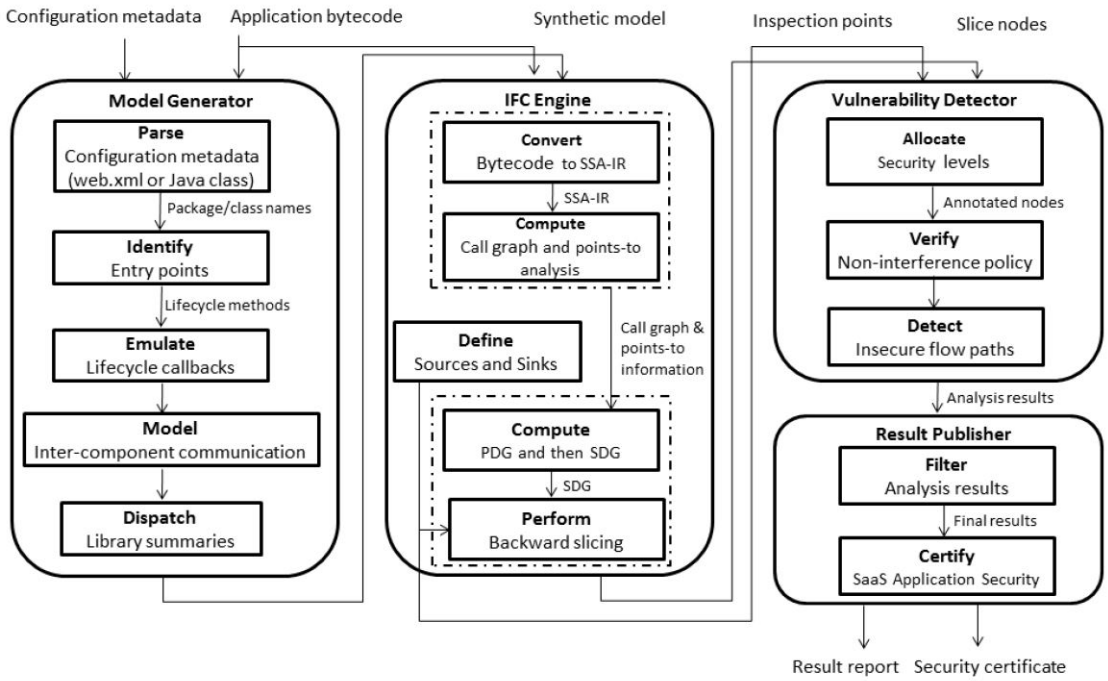

Fig. 2. Service Layer Design Architecture Presented in [38].

IRM system, thus, creating their own IFC system and enforcing security in this untrusted cloud infrastructure. This kind of system can be very useful for banks or similar institutions.

H. Chinese Wall - General Implementation

Chinese Wall Security Policy [42] makes use of subjects and objects to prevent information flows which cause conflict-of-interests between tenants. Data is divided into conflict categories (conflict-of-interest class) and subdivided into sub-divisions based on their profile. When data from a sub-division is accessed, all data from that sub-division can be accessed but data from the broad conflict class can't be accessed anymore but data from other broad conflict classes can be. Access to data is constrained by what data the subject has already accessed. All subjects are allowed to access at most one data-set which belongs to a same conflict-of-interest class. A subject can freely access any object in the sanitized security group, which is a conflict category.(data which doesn't need restriction). [43] For scaling purposes we could use a decentralised Chinese wall mechanism mentioned by Minky. [44] The decentralised mechanism uses Law Governed Interaction mechanism where authorization is required before accessing any resource. Chinese-Wall Process Confinement (CWPC) could be used for practical application-level distributed coalitions that provide fine-grained access controls for resources and that emphasize minimizing the impact on the usability [45]. The centralized system can be applied to serverless systems by classifying the functions using conflict categories and using that strategy to allocate VM's. Thus, there is lesser conflict of functions on the same VM. The CWPC way can be used to check for data request access on the functions from the service provider and the validation method which can be used based on the Minky Law Governed Interaction way where functions have to authorize the usage of data by the other functions if there is an conflict. This is an idea of to increase trust between functions and their working together with conflicts. This system requires a lot of trust by the functions to the cloud provider and the cloud provider needs to maintain strict security policies so that no information gets leaked as the cloud provider will have a lot of sensitive information with this method.

VIII. Advantages

DIFC [46] will be useful on serverless systems for managing and securing information flows both within and between virtual machines and, the overall flow within the cloud. Continuous check of data at every usage point will prevent data leakage and unauthorized use. It would also allow the applications to define their own independent security terminology dynamically [1]. IFC tracks all data flows in order to detect policy violations, it can be used to provide detailed logs for audit purposes [1]. The IRM system makes it possible for systems like banks to switch to public cloud, still having and enforcing their own high level of security. The warm container method increases confidence and decreases security vulnerability with an IFC implementation.

IX. Trade-off

- Data sharing between virtual machines could be done through the intranet having IFC enabled on the network level of the cloud system or have the data sourced through a secure system where it gets authorized. Using a secure system for authorization would reduce the burden on the developer but will only leave a generic check mechanism on IFC and possibly a higher overhead.

- There is a higher development and design time for DIFC integration which can help reduce the security vulnerabilities for the systems integrating themselves with the security policy.
- Implementing a DIFC model at different levels in the cloud system where it is integrated with the other levels and set by the cloud provider, one of the main advantages could be the network analysis which restricts the flow of sensitive data outside the cloud intranet. We can also restrict the flow of data between functions with labelling but the problem here would be that on a real world scale which have billions of functions and a very vast and diverse infrastructure, a lot of labels would be required for some part of data from a label to pass to another and restrict to others/some specific ones.
- Flow of data received by a function can be blocked by blocking transitivity, but it might break the functionality if it is required, so transitivity is also a parameter that would have to be considered with labels. With a vast network, the checking of this data at every point would be a heavy overhead apart from managing the data and where it could flow. This would in turn increase the development time and one would have to continue checking till it reaches the end point, checking all allowed use cases, adding extra labels which are allowed to use its data and adding restrictions based on when that data flow is supposed to stop.
- Updating the infrastructure or functions would affect each other and has a very high chance of breaking functionalities, thus, upgrading would have to be in phases where both phases are working at a time till everyone moves to the new model. So the effort for updating is increased as there can't be fast updates and any update would force every function that it depends on and those functions that depend on it to be updated and tested.
- Onboarding of new personal would take a lot of time to this system.
- Taint tracking systems would not work if the developer tried to evade them. Using channels outside of the policy are known as covert channels [47].
- Setting up a system like this for a cloud provider would take a lot of effort and money, which wouldn't necessarily result in an increase in revenue but would lead to more confidence by the tenants in the cloud system. Tenants setting up their security from a cloud provider they don't trust would require very heavy security and is almost impossible without set models and to only setup for one particular application. Thus, the feasibility of this system is forced on people making a generic model which is adapted and updated by the tenants and cloud providers like other services.
- The cloud provider will have a lot more sensitive and analyzed data because of this system, thus, the cloud provider keeping this data secure and not exploiting this data would require legal implications so as to keep the cloud provider in check and regular audits by a central authority which regularly audits the cloud system and the data used for the cloud auditing and confidence that it hasn't been tampered with can be gotten by this(DIFC) system itself.

X. Implementation

A nontermination sensitive DIFC can be setup on the cloud which is integrated into the network level, OS level, only allowing authorized system calls checked by the cloud provider and the specific tenant receiving the request based on their request policies. Using the VM level implementation mentioned above securing itself from the bare-metal machine attacks. A general norm of label-set which are configured in the cloud system and could be extended by the application, along with an intranet DMZ set which blocks the flow of data outside the DMZ unless it contains authorized labels. DIFC having library extensions(eg boto3 by amazon for AWS for python language) which be imported and extended to the applications. A system like this could have checks from the starting point i.e the REST call till the end of point where the data will flow and could also have legal limitations which check the data based on location among others. All third-party plugins into each function are used with end-to-end integrity protection and the functions can define policies so that selected plugins never interact and, potentially corrupt sensitive data or only certain plugins interact with sensitive data. A Chinese wall setup could be used to present and restrict any functions being run on the same machine which have conflicts with each other. There could be a service level implementation where any suspected activity could be checked before sending it to the actual function, as this would have a high overhead but the payment of this system would have to be figured out. Any function which wants extra security could have sticky policies which could be used to log all data decryption, thus always having a log of everybody getting the data at this source and the data is only present here and only gets decrypted through the sticky policy. The overall flow of data is monitored using the IFC system and logged and thus, these detailed logs can be used for verification and checking of the cloud infrastructure, tenants and other functions. The logging level shown will only be for their data and the other data will be obliqued. The system will have multiple data copies for different labels to stop implicit storage channels and with function owner will be notified if any activity which does not follow the policy defined for the storage channels. Warm Startup is present with all sensitive data being removed which was identified by tainting. This system would improve security confidence between functions and also with the cloud provider. Blockchain methodology could be added to the Audit log to show that it hasn't been modified and present confidence in the Audit Log.

XI. Future Research Direction

- A Dynamic IFC Model for serverless systems is made by [21] in which the main points are described above in the Network Level Section. The model has inherent assumptions where almost all of the TCB is considered safe and data integrity isn't considered. Reuse of containers and warm starts or multiple invocations to the same function aren't considered. The system could be extended to include all these points.

- With serverless systems a lot of the services required are outsourced, for example the authentication service for AWS i.e AWS Cognito is used in a lot of serverless functions rather than using their own. What kind of extra risk does outsourcing things bring and how does IFC mitigate it is another field where more information with some data for proof is needed, the ideas are expressed in this paper.
- The monitoring of function requests moves the stateless architecture to a stateful architecture. How could we implement IFC while keeping a stateless architecture.
- A computation layer based on label, isolated in a DMZ where only the public output is allowed to leave the overall virtual DMZ network, so all private computation is done inside.
- An cloud implementation which presents the model with an norm based label structure and doesn't have a heavy overhead on performance and time compared to the present system. A prototype proof of concept for the real world.
- An implementation of a model described above could be a starting point with all the system design features mentioned considered. The implementation shouldn't consider TCB to be safe.

XII. Conclusion

A survey of the present IFC implementations is presented and system designs which are relevant to Serverless Systems that could be added to Serverless Systems Architecture and, an idea of an IFC model that could be effectively applied in a decentralised model like serverless systems. The overall idea of this paper assumes that all of the TCB is vulnerable and gives implementations/ideas which could be used to increase confidence between functions with other functions and cloud provider and also mitigate security vulnerabilities making the system safer.

Acknowledgment

I thank Danfeng Zhang for his helpful suggestions on early drafts of the paper.

List of Abbreviations

ACL: Access Control List
DIFC: Decentralized Information Flow Control
DMZ: Demilitazied Zone
IFC: Information Flow Control
RBAC: Role-Based Access Control
TA: Trusted Authority
TCB: Trusted Computing Base
TNSI: Termination-Sensitive Non-interference
VM: Virtual Machine

References

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

[2] Alexander Posashenki. How serverless is chanding security: The good, bad, ugly, and how to fix it. https://distillery.com/blog/serverless-is-changing-security/, 2019.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 419–434, 2020.

[4] Tim Wagner. Understanding container reuse in aws lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2014.

[5] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.

[6] Kostya Kortchinsky. Cloudburst: A vmware guest to host escape story. *Black Hat USA*, 19, 2009.

[7] Nelson Elhage. Virtunoid: A kvm guest- host privilege escalation exploit. *Black Hat USA*, 2011, 2011.

[8] Dirk Leinenbacih and Thomas Santen. Verifying the microsoft hyper-v hypervisor with vcc. In *International Symposium on Formal Methods*, pages 806–809. Springer, 2009.

[9] Rich Jones. Gone in 60 milliseconds intrusion and exfiltration in server-less architectures. https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds/, 2016.

[10] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[11] Avraham Shulman, Ory Segal, and Shaked Yosef Zin. Methods for securing serverless functions, January 3 2019. US Patent App. 16/024,863.

[12] Develop fast, stay secure. https://snyk.io/.

[13] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms. *International World Wide Web Conference Committee (IW3C2)*, 2020.

[14] Johannes Manner, Stefan Kolb, and Guido Wirtz. Troubleshooting serverless functions: a combined monitoring and debugging approach. *SICS Software-Intensive Cyber-Physical Systems*, 34(2-3):99–104, 2019.

[15] Jeremy Daly. Event injection: Protecting your serverless applications. https://www.jeremydaly.com/event-injection-protecting-your-serverless-applications/, 2018.

[16] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication and access control vulnerabilities in web applications. 2009.

[17] Thomas Pasquier, Jean Bacon, Jatinder Singh, and David Eyers. Data-centric access control for cloud computing. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, pages 81–88, 2016.

[18] Thomas FJ-M Pasquier, Jatinder Singh, Jean Bacon, and Olivier Hermant. An information flow control model for the cloud.

[19] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[20] Joseph A Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE, 1982.

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

[22] Hayawardh Vijayakumar, Guruprasad Jakka, Sandra Rueda, Joshua Schiffman, and Trent Jaeger. Integrity walls: Finding attack surfaces from mandatory access control policies. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 75–76, 2012.

[23] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-order and symbolic computation*, 14(1):59–91, 2001.

[24] Nevin Heintze and Jon G Riecke. The slam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, 1998.

[25] Siani Pearson and Marco Casassa-Mont. Sticky policies: An approach for managing privacy across multiple parties. *Computer*, 44(9):60–68, 2011.

[26] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269. IEEE, 2005.

[27] Thomas H Austin, Tommy Schmitz, and Cormac Flanagan. Multiple facets for dynamic information flow with exceptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(3):1–56, 2017.

[28] Thomas H Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–12, 2010.

[29] Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 165–178, 2012.

[30] Thomas FJ-M Pasquier and David Eyers. Information flow audit for transparency and compliance in the handling of personal data. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, pages 112–117. IEEE, 2016.

[31] Afshar Ganjali and David Lie. Auditing cloud management using information flow tracking. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 79–84, 2012.

[32] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A Reis, Manish Vachharajani, and David I August. Rifle: An architectural framework for user-centric information-flow security. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 243–254. IEEE, 2004.

[33] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 39(11):85–96, 2004.

[34] Thomas FJM Pasquier, Jean Bacon, and David Eyers. Flowk: Information flow control for the cloud. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 70–77. IEEE, 2014.

[35] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4):15–27, 2006.

[36] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. *ACM SIGOPS Operating Systems Review*, 41(6):321–334, 2007.

[37] Jatinder Singh, Thomas FJ-M Pasquier, Jean Bacon, and David Eyers. Integrating messaging middleware and information flow control. In *2015 IEEE International Conference on Cloud Engineering*, pages 54–59. IEEE, 2015.

[38] Marwa Elsayed and Mohammad Zulkernine. Ifcaas: information flow control as a service for cloud security. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 211–216. IEEE, 2016.

[39] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[40] Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[41] Safwan Mahmud Khan, Kevin W Hamlen, and Murat Kantarcioglu. Silver lining: Enforcing secure information flow at the cloud edge. In *2014 IEEE International Conference on Cloud Engineering*, pages 37–46. IEEE, 2014.

[42] David FC Brewer and Micheal J Nash. The chinese wall security policy. In *null*, page 206. IEEE, 1989.

[43] Ruoyu Wu, Gail-Joon Ahn, Hongxin Hu, and Mukesh Singhal. Information flow control in cloud computing. In *6th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2010)*, pages 1–7. IEEE, 2010.

[44] Naftaly H Minsky. A decentralized treatment of a highly distributed chinese-wall policy. In *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.*, pages 181–184. IEEE, 2004.

[45] Yasuharu Katsuno, Yuji Watanabe, Sanehiro Furuichi, and Michiharu Kudo. Chinese-wall process confinement for practical distributed coalitions. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 225–234, 2007.

[46] Andrew C Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[47] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *2011 IEEE Symposium on Security and Privacy*, pages 413–428. IEEE, 2011.