FUSE: Fault Diagnosis and Suppression with eBPF for Microservices

Gowri Sankar Ramachandran(✉), Lewyn McDonald, and Raja Jurdak

Trusted Networks Lab, Queensland University of Technology, Brisbane, Australia
g.ramachandran@qut.edu.au

Abstract. Contemporary applications harness microservices architecture to attain scalability, loose coupling, and abstraction advantages. This approach involves breaking down applications into smaller, composable services, which are hosted in the cloud. Cloud deployment offers advantages like elastic load balancing, cost-efficiency, and ease of management. However, it raises two issues: trusting third-party providers and limited fault diagnosis due to generic logs. Deep runtime introspection of microservices on third-party clouds can enhance the resilience of cloud-native microservice-based applications.

This paper introduces *FUSE*, a novel framework based on eBPF technology that enables deep introspection of microservices' runtime behavior. FUSE observes microservices at the kernel level, tracing system calls, function invocations, and disk accesses to create a unique hash-based digest for each microservice invocation. This digest is then used to verify runtime correctness: correct microservices consistently produce a known, deterministic digest, while faulty services generate random traces. FUSE provides real-time fault detection and suppression, preventing cascading failures. Additionally, it introduces a *stability* score for succinctly capturing runtime consistencies in microservices. In our evaluation with four representative microservices on AWS EC2 instances, FUSE successfully detected 53 *runtime faults.*

Keywords: eBPF · Resilient Microservice · Fault Diagnosis

1 Introduction

Contemporary applications adopt microservices architecture to achieve scalability, load balancing, continuous integration and loose coupling for large-scale enterprise applications [7]. Some of the largest tech companies, including Amazon [8] and Microsoft [12], serve millions of customers following microservices architecture. The emergence of cloud computing platforms further accelerates the growth of microservices, allowing the service owners to deploy their applications on the cloud without needing to invest in hardware, software, and tooling resources, as existing cloud platforms offer many built-in services, including elastic load balancing, security, and remote management, for an affordable cost, making them attractive for the deployment of microservices [21].

F. Monti et al. (Eds.): ICSOC 2023, LNCS 14419, pp. 243–257, 2023.
https://doi.org/10.1007/978-3-031-48421-6_17

As the applications switch from the "monolithic" to the "microservices" model, the end system consists of a set of independent services running on different cloud instances in a virtualised environment. Although cloud providers strive to provide reliable services to the service owners, there is still a possibility of service failures and faults at runtime [1,5,6]. Service meshes have been developed to make microservices resilient against communication failures, as the failure of a single service could impact other inter-connected services through the cascading effect [11]. Figure 1 (left) shows how a single faulty service can disrupt other interconnected services due to fault propagation. Sidecars monitor individual services and relay the information to the service mesh control plane to make dynamic decisions at runtime. Existing features of sidecars include traffic monitoring, load balancing, and fault-tolerant networking, but there is no support to *deeply* observe the "execution" of microservice at runtime to ensure consistency [4,14].

Existing literature highlights the benefits of observing microservices at the kernel level using the extended Berkeley Packet Filter (eBPF) [3,4,10,13,18,22]. eBPF offers a rich set of functionalities to monitor the runtime behaviour of microservices. It enables the service owners to develop *kernel probes*, which are user-defined programs with kernel privileges to deeply introspect the behaviour of applications running in the user space. eBPF provides several functionalities, including support for network observation, disk monitoring, and system call tracing to get fine-grained activity logs of user-level programs such as microservices. Extant literature proposes eBPF-based frameworks to classify microservices in data centres [3,4] and for monitoring network and performance [13,18,22], lacking a solution to detect and suppress runtime faults in microservice-based applications, which is the focus of this work.

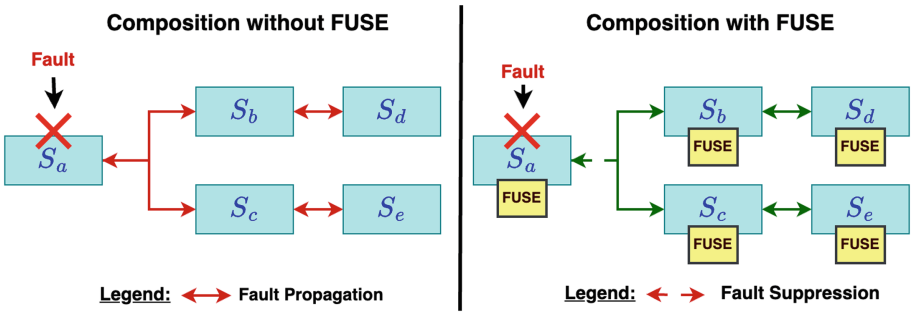

Fig. 1. Composing Microservices using FUSE to suppress fault.

This paper introduces *FUSE*, an eBPF-based fault-diagnosis and suppression framework for microservices architecture. FUSE differs from other works in the following ways:

- FUSE observes the runtime behaviour of microservices by monitoring system calls, function invocations, and disk accesses, including memory allocations, to create a *unique signature* (or digest) from traces for each invocation.
- FUSE categories microservices as an idempotent or non-idempotent service based on the signature.
- FUSE detects runtime faults by checking the signature of each invocation against the expected signatures. Upon fault detection, it promptly alerts the service administrator and the circuit-breaker to suppress cascading failures, as shown in Fig. 1 (right).
- FUSE also helps the service owners and developers quantify the runtime consistency of microservices through a novel *stability score* for microservices. A microservice composition with a high stability score is good for resiliency.

FUSE is implemented in a Linux environment with eBPF and evaluated on AWS EC2 instances running the Ubuntu operating system. The performance and effectiveness of FUSE are validated using four representative microservices that 1) register a user into a MySQL database, 2) query the system to retrieve the list of users, 3) test the strength of a password, and 4) add the numbers in a list. During the evaluation, FUSE successfully detected 53 runtime faults, highlighting its ability to prevent cascading failures by alerting service administrators and circuit breakers. Lastly, the stability score is a valuable feature for service administrators and software developers to improve the microservices' resiliency.

2 Background

2.1 extended Berkeley Packet Filter (eBPF)

The extended Berkeley Packet Filter, commonly known as eBPF, is a special type of virtual machine within the Linux kernel [15]. It was introduced in 1992 with a register-based filter for evaluating the network packets deep inside the kernel. *tcpdump* is one of the popular tools from the eBPF family. Although network monitoring is one of the popular use cases of eBPF, it does provide multiple hooks to trace and monitor various subsystems inside the Linux kernel. Some of the other hooks include Filetop (for tracking the file reads and writes), Opensnoop (attempts to track the files accessed by a specific process), and syscount (counts the number of system calls) [9]. The user-level programs can attach to these kernel hooks through eBPF probes such as kprobe, uprobe, tracepoint, and socket [9]. Depending on the type of selected hooks, the probes deliver detailed information to the user for deep introspection of application activities, including microservices.

2.2 Faults in Microservices

Microservices use a software framework such as Apache and Spring and run on a hardware infrastructure provided by the cloud provider in the case of cloud-native deployments. As discussed in [1], the microservices could be exposed to

intermittent hardware faults due to cosmic radiation or impure packaging material. Facebook's data centres that serve a multitude of apps, such as Facebook, WhatsApp, and Instagram, experienced *silent data corruptions*, causing inaccurate computations [6]. Such runtime silent faults occur as a result of manufacturing defects at the silicon level [5]. The aging of microservices also introduces faults, which are undetected by Kubernetes probes, according to [17]. BROFY [10] explains the need to develop approaches for microservices' integrity validation as the attackers or faulty hardware could introduce *bitflip errors* (one or more of the bits gets flipped, changing the runtime behaviour) to the computation infrastructure, causing silent and undetectable failures. Additionally, services may fail to access the desired resource, including the database, to fulfil the functional requirements, resulting in run-time faults. These problems underscore the importance of developing approaches to detect runtime faults in microservices, which is the focus of this work.

3 Related Work

Existing works have studied approaches to enhance the monitoring capabilities of microservices to detect performance issues [3,4,13,18,22]. These works leverage eBPF to observe microservices' runtime behaviour by focusing on the networking activities [13], including TCP traffic [22] and latency [18], CPU activations [3, 4,22], Block I/O performance [22] to understand the performance bottlenecks. Although these works aim to improve microservices' performance, they don't propose fault diagnosis or suppression.

MAGNet [3] is similar to FUSE in the aspect of application-focused eBPF tracing, but it aims to generate identities for workloads in data centres without detecting faults using the eBPF traces. Hyunseok *et al.* [4] introduced a microservices fingerprinting technique by tracing the system calls used by microservices and trained a machine learning model to classify services. FUSE classifies the microservices based on eBPF trace, but it uses system calls, disk I/O activities, and function invocations to generate a unique digest per invocation without employing machine learning. Furthermore, FUSE contributes a fault detection and suppression framework and a stability scoring mechanism to tackle runtime faults.

The stability of microservices is studied using eBPF in [20], which leverages variable autoencoders to detect unstable or compromised containers based on eBPF traces. It monitors a pre-configured set of 72 Linux system calls to capture specific security incidents and application faults. This work is similar to ours in the aspect of the eBPF-based approach for stability analysis, but we focus on application-level or microservices fingerprinting using system calls, function invocation, and disk accesses without involving any machine learning. Areeg and Claus [19] detect anomalies in containerised microservices using Markov Models, wherein the key performance indicators such as *mpstat* and *vmstat* are collected at the application level to learn a model using Hierarchical Hidden Markov Models, which detect abnormal microservice behaviours with 97% accuracy. Unlike

this paper, FUSE generates microservice-specific eBPF traces and detects stability and abnormality through a hash-based signature without applying machine learning algorithms. Many other works tackle anomalies or eBPF-based observability for microservices, but they don't focus on fault detection using eBPF-based tracing by combining system calls, function invocations, and disk accesses. Besides, none of the existing works introduces a stability scorecard for microservices, which is another novel contribution of our work.

4 FUSE: Fault Diagnosis and Suppression with eBPF for Microservices

4.1 System Model

Modern applications are composed of interconnected microservices, wherein each service $s_x \in \mathcal{S}$ could be connected with one or more services. In such circumstances, whenever the end user issues a request to a service, it may trigger a series of services to generate a response to the user. Upon receiving a request req_x, the service s_x processes the request by running computations exe_x and produces a response res_x. Here, exe_x may involve contacting other services, meaning the response could be generated with the help of other dependent services. Each service in $\mathcal{S}$ gets invoked numerous times in deployment, depending on the application's popularity and the customer base. Each service gets invoked $\mathcal{K}$ ($\mathcal{K} \in \mathbb{N}$) times in deployment, and the i^{th} invocation of a service s_x can be denoted by I_x^i. Each invocation includes the execution phase (exe_x^i), which will use the hardware and software resources, including the Linux kernel. eBPF provides tools to introspect the behaviour of a service s_x at the execution phase (also called "runtime") by capturing kernel-level traces for exe_x^i, which is denoted by $trace_x^i$. In summary, i^{th} invocation of a service s_x can be represented by a tuple $\langle req_x^i, exe_x^i, res_x^i, trace_x^i \rangle$. This work builds on some key concepts, such as idempotency and stability, which are defined below.

Definition 1. ***Idempotency*** *The system's state remains the same when an operation is executed any number of times [16].*

GET and *HEAD* are examples of idempotent HTTP operations because they don't change the server's state on the request' successful completion. Besides, even the same request can be issued many times without altering the server's state. On the other hand, the *POST* operation of HTTP changes the server's state, meaning it may add a new item to a database; hence, it is *non-idempotent.*

Definition 2. ***Idempotent Microservice*** *A microservice is considered idempotent if it doesn't alter the system's state when a request is processed any number of times.*

A microservice that reads an employee's data from a database is an example of an idempotent microservice. In contrast, the service that registers a new employee's data is non-idempotent as it changes the server's state.

Definition 3. ***Trace*** *A trace of a microservice corresponds to the* low-level activities *that the service carries out inside the kernel to process a request* req_x. $\mathcal{T}_x$ *denotes a set of* ***unique*** *traces of service* s_x *and* $\mathcal{L}_x$ *denotes the length of* $\mathcal{T}_x$.

Software applications access the CPU, memory, and network by invoking schedulers, I/O management modules, and network managers to fulfil the desired functionalities. Here, the low-level activities correspond to all activities that happen within the kernel as part of the application's or microservice's execution. This work assumes microservices perform single functions upon receiving requests (req_x) with deterministic, bounded input sizes. For instance, the *register user to a database* service limits 'user name' input to 20 characters.

Definition 4. ***Idempotent Trace*** *A microservice's trace is idempotent if it doesn't change when the same or different requests are processed any number of times.*

Considering three invocations, p, q, and r, of service s_x the trace is said to be idempotent if and only if:

$$trace_x^p = trace_x^q = trace_x^r = \alpha, \ \ where\ \mathcal{T}_x = \{\alpha\}\ and\ \mathcal{L}_x = 1 \tag{1}$$

Definition 5. ***Stability of a Microservice:*** *A microservice is said to be stable if its traces are idempotent for a given request req. The stability of a service in percentage is calculated using traces generated from* $\mathcal{E}$ *invocations. The stability of a microservice,* sm_x*, is:*

$$sm_x^{req} = (1 \div \mathcal{L}_x) * 100 \tag{2}$$

Figure 2 (top) shows an example of an idempotent service with a stability score of 100% for $\mathcal{E}$ of 4 because it produces one unique trace. But, some services may produce more than one trace for $\mathcal{E}$ ($\mathcal{E} \in \mathbb{N}$) invocations. Figure 2 (bottom) illustrates an idempotent service with a stability score of 50% for $\mathcal{E}$ of 4, with two unique traces α and β. The higher stability score indicates runtime consistency, helping developers and administrators to build highly stable services.

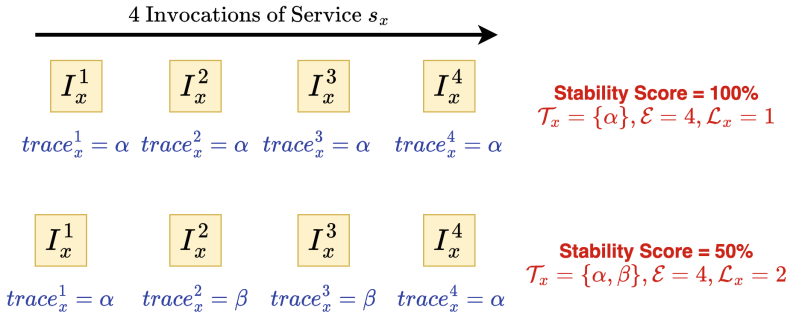

Fig. 2. Stability Score of Idempotent Service based on 4 Invocations.

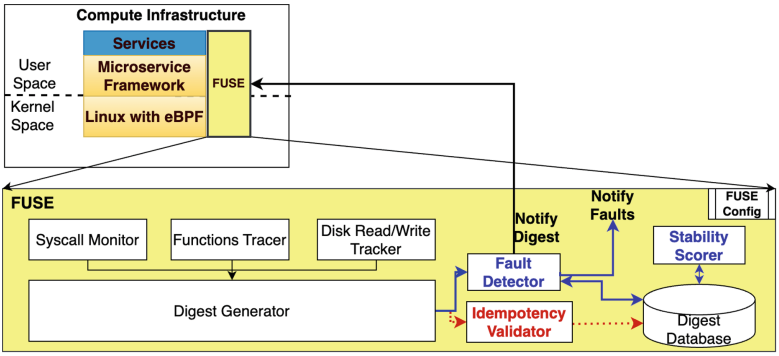

Fig. 3. FUSE Architecture: The dotted line indicates operations in idempotency validation mode, while normal line denotes fault detection and suppression operations.

4.2 Architecture of FUSE

At a high level, FUSE is a cross-layer architecture running at user and kernel spaces, as shown in Fig. 3. Microservices run in a computing environment, including a cloud instance. Typically, services are hosted on a framework such as Spring Boot. An operating system such as Ubuntu manages the resources for the application-level services. In this model, the operating system functions within the kernel space without offering much visibility to the users. In contrast, the services sit in the user space, leveraging the operating system to function correctly. FUSE leverages eBPF to monitor and extract kernel space activities of services to generate signatures and suppress faults. The user space activities for eBPF include setting up the necessary hooks and the configurations to generate, process, store, and validate traces. In contrast, the kernel space intercepts the traces required for the target microservices. The fundamental building blocks of FUSE are discussed in the rest of the section.

FUSE Configurations. FUSE can operate in two modes, including *idempotency validation* and *fault detection and suppression* modes, based on the configurations set by the microservices administrator. In the *idempotency validation mode*, FUSE runs the same microservice $\mathcal{E}$ times and collects the traces in a database. Subsequently, the traces are analysed to determine whether the microservice is *idempotent or not*. This mode must be activated for each microservice before production for fault diagnosis and suppression. The *fault detection and suppression mode* is activated at runtime to validate the microservices' trace for each invocation against the database. As shown in Fig. 3, this mode uses a different set of building blocks for detecting and suppressing faults based on the runtime traces.

Syscall Monitor tracks the system calls used by the microservice. At each microservice's invocation, this building block intercepts the system calls and generates a list, including the counts for each system call. For example, read is one of

Algorithm 1: FUSE in idempotency validation and fault detection and suppression modes: Stability score calculation is included in idempotency validation mode.

FUSE in idempotency validation mode

Require: S_x, $\mathcal{E}$
$\mathcal{T} \leftarrow \{\}$ */* Empty $\mathcal{T}$ */*

for i = 1 to $\mathcal{E}$ **do**
$trace_i$ = generateDigest(S_x)
if $trace_i \notin \mathcal{T}$ **then**
$\mathcal{T} = \mathcal{T} \cup trace_i$
end if
end for
storeTraceinDatabase(S_x,$\mathcal{T}$)
return

FUSE Stability Score Calculation

Require: S_x, $\mathcal{T}$
$\mathcal{L} \leftarrow 0$ */* Length of $\mathcal{T}$ is 0 */*
$sm \leftarrow 0$ */* Stability score is initialised 0 */*
$\mathcal{L} = |\mathcal{T}|$ */* Length of $\mathcal{T}$ */*
$sm = (1 \div \mathcal{L})$*100 */* Stability score is calculated */*
storeStabilityScoreinDatabase(S_x,sm)
return

FUSE in fault detection and suppression mode

$trace_j$ = generateDigest(S_x) */* j^{th} invocation of a service */*
if $trace_j \notin \mathcal{T}$ **then**
notifyFault()
end if

the widespread system calls, and it opens a file or a resource. **Functions Tracer** traces the functions and libraries invoked by the microservice. For each invocation, this captures the list of files and libraries accessed by the microservice. **Disk Read/Write Tracker** tracks the number of reads and writes, including memory allocations, that happen during the execution of microservice along with a pointer to a file, indicating the files that were read from or written to.

Digest (or Signature) Generator processes the traces generated by the syscall monitor, functions tracer, and disk read/write tracker to create a digest or signature for each invocation. The digest is made by counting the number of calls, the list of functions invoked, the amount of data read from or written to, and *malloc* allocations. A cryptographic hash function such as sha256 creates a unique signature per invocation.

Idempotency Validator processes the digests during the *idempotency validation mode* by studying the consistency of digests. Each microservice is executed $\mathcal{E}$ times to perform idempotency validation. Therefore, this process is data intensive as multiple traces must be generated to determine whether a given microservice is *idempotent* or not. An idempotent microservice would always produce the known cryptographic hash for a given request type req_x because such a microservice always executes the same system calls, accesses the same list of functions, has the same data read from or written to, and allocates the same amount of memory. Algorithm 1 explains FUSE's operations.

Digest Database stores and manages digests associated with a microservice-based application. It is important to note that each microservice need not have its database; instead, a single database can be used to manage the digests.

Fault Detector gets activated in *fault detection and suppression mode*, and it is responsible for checking whether a digest of a recent microservice's invocation produces the known and expected digests. Recall that an idempotent microservice will have the known digest unless there is a fault. This module checks the digest and generates an alert to the service admin. Besides, the digest is also notified to the circuit breaker for fault suppression.

4.3 Fault Diagnosis and Suppression

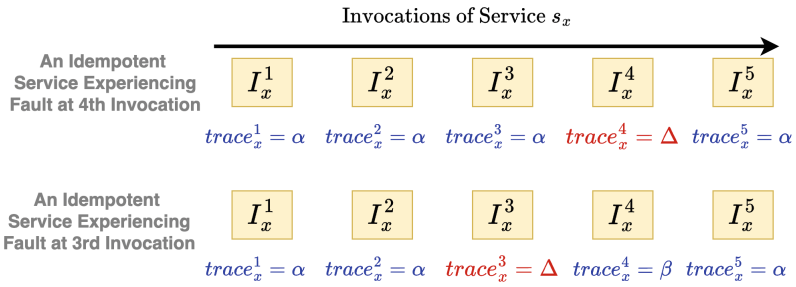

Fig. 4. Fault breaks the idempotency of an idempotent service.

Definition 6. ***Faulty Microservice*** *A microservice is said to be faulty when its traces are* ***intermittently*** *non-idempotent. An idempotent microservice can experience faults at runtime, breaking idempotency.*

Figure 4 elucidates how an idempotent microservice can experience an unexpected fault at runtime due to hardware or software failures, breaking the trace consistency. Note that any trace other than α is considered faulty for the first idempotent service in Fig. 4 (top). On the other hand, Fig. 4 (bottom) shows an idempotent service with more than one known trace to be considered non-faulty;

however, the 3rd invocation (see) produces an unexpected trace of Δ, denoting a runtime fault.

Figure 1 (left) shows that a single faulty service could impact other interconnected services if left untreated. Circuit breakers are recommended to increase the resiliency and availability of microservices in the event of cascading failures. A circuit breaker relies on *fault event* to open the circuit to prevent cascading failures. Network-related issues, server failures, and overloads are considered fault events in circuit breakers, allowing microservices to overcome major and obvious faults. Our approach complements existing techniques and further strengthens the circuit breakers by tackling less-obvious faults, which could arise due to hardware abnormalities or any unexpected behaviour only noticeable at runtime, including the involvement of an adversary. Our fault suppression technique is presented in the next section.

4.4 Fault Suppression

FUSE digests (or signatures) provide a stable reference for fault diagnosis, as a service could be *idempotent or non-idempotent*. Our fault suppression technique is proxy-based [2], meaning that the service interacts with other dependent services if and only if the digest of the current invocation satisfies the idempotency property (see Definitions 4). Any variations to the digest (or trace) indicate a fault (see Definition 6), suppressing outbound communications with dependent services, as shown in Fig. 1 (right). The circuit breaker opens the connection to prevent cascading failures. The *fault detection and suppression* mode monitors the trace following Algorithm 1 and notifies faults to the appropriate agent. Following a proxy-based fault suppression scheme, the outgoing requests to dependent microservices are blocked to prevent cascading failures, as shown in Fig. 1.

4.5 Stability Score

FUSE validates the stability of microservices through a stability scorer module, as shown in Fig. 3. As discussed in Definition 5, the stability of a microservice depends on its trace (or signature) consistency. Our stability scoring mechanism provides a score between 0% and 100% for microservices. A score closer to 100% indicates high stability, while any score close to 0% indicates poor stability. An idempotent service has a high stability score, while non-idempotent services have a low stability score. The stability scorer module takes the digests from the database and counts the unique digests per microservice in the *idempotency validation mode*, as shown in Algorithm 1.

This score is beneficial for system administrators and microservice architects. From the service management and resilience viewpoint, having an idempotent microservice with a stability score of 100% maximises the determinism and fault detection capabilities. In contrast, any score close to 0% shows the non-determinism of the microservice. Besides, when designing a microservice, the developers can aim to compose a *strictly idempotent microservice* by using

FUSE's stability score as it helps the developers understand hidden uncertainties in the code, which is only noticeable at runtime. Figure 2 shows how the stability score is calculated for an idempotent microservice.

5 Proof-of-Concept Implementation and Evaluation

FUSE is implemented using eBPF and tested on AWS EC2 instances running the Ubuntu operating system. To validate the practicality of FUSE, we have developed POST and GET services using Python Flask with the following functionalities: **1) New user registration service (S1):** receives a POST request with the user details, including the name, address, and country, and stores them in a *MySQL* database. **2) Users data retriever service (S2):** handles a GET request to gather the users' data from a *MySQL* database and sends it back to the requester. **3) Password strength checker service (S3):** receives a POST request to check the strength of a password the user selects. It receives a password string and checks its strength by assessing the lower and upper cases, digits, and length, and returns the password strength as *Strong* or *Weak*. **4) Addition service (S4):** adds the numbers in the request and returns the sum as a response. Our proof-of-concept implementation used a proxy that generates and validates the trace (or signature) for each POST or GET request. We evaluate the performance and stability of the example representative services and report the results. Each service was executed more than 1000 times in *idempotency validation* mode, i.e. $\mathcal{E} > 1000$.

5.1 Idempotency of Example Services

For $\mathcal{E} > 1000$, services S1, S2, S3, and S4 have $\mathcal{T}$ of 2, 4, 3, and 10, respectively, of which a single trace is dominant, making them idempotent. The sha256 hash is shortened to four characters in Table 1 for brevity. However, the real digest is 64 characters long. The amount of memory used, the number of functions accessed, and system calls invoked changes depend on the microservices, resulting in unique hashes per service. Figure 5 shows the distribution of hashes for S1, S2, S3, and S4 - it is clear that each microservice has a dominating hash that appears more than 97% of the time. Besides, S1, S2, S3, and S4 have stability scores of 50, 25, 33.3, and 10, respectively. The higher stability score corresponds to high determinism and stability. S1 is the most stable among the example services, while S4 is the least stable. Figure 6 shows how the stability score evolved with each invocation and stabilised. The instability of S4 and the high stability of S1 are apparent in Fig. 6.

Table 1. Stability Score of Services based on Traces.

Service	$\mathcal{T}$	$\mathcal{L}$	$\mathcal{E}$	Stability Score (in %)
S1	{'8507':7, 'f8ec':998}	2	1005	50
S2	{'3173':2, '7eb0':1015, '913b':1, '51a7':4}	4	1022	25
S3	{'075d':1094, '4532':11, '6947':1}	3	1106	33.3
S4	{'05db':1, '23e2':3, '4d6b':6, '6398':1163, '68c0':1, '735d':4, '8801':2, 'e280':2, 'f541':7, 'd447':1}	10	1190	10

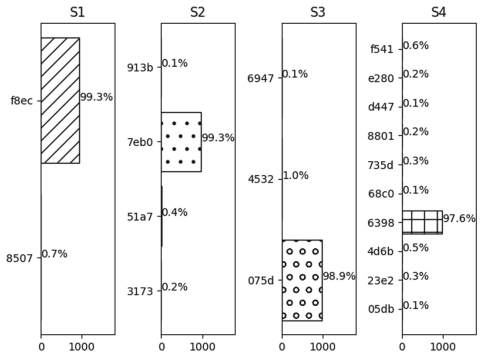

Fig. 5. Distribution of FUSE Traces for Services S1, S2, S3, and S4.

5.2 Overhead of FUSE

Traditional microservices don't rely on kernel-level traces for fault suppression. Thus, they don't incur any overhead within the request-response cycle. When a request is sent to a microservice, it gets processed, and the response is sent back. However, FUSE introduces an overhead in storage and latency. The **Storage Overhead of FUSE** originates from the storage of traces generated. FUSE produces multiple eBPF trace files to store file accesses, system call statistics, memory allocations, and disk operations. The storage overhead of FUSE per microservice execution is presented in Table 2, wherein the disk IO trace file takes up the most space while the system calls take up the least space, but these can be deleted periodically or immediately based on the requirements.

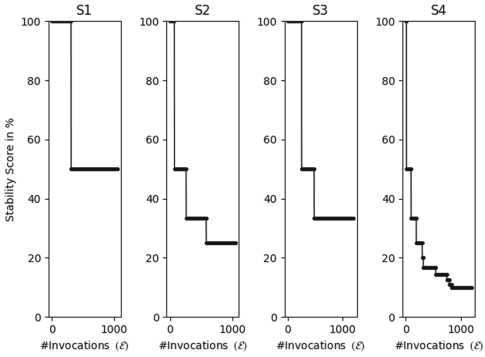

Fig. 6. Stability Score vs. Number of Invocations for S1, S2, S3, and S4.

Table 2. Storage Overhead for Trace Files

Service	Syscall Trace (Bytes)	Function Trace (Bytes)	Disk I/O Trace (Bytes)
S1	751	1710	16380
S2	751	1710	16379
S3	751	1647	16380
S4	1039	1647	16385

The **Latency of FUSE** differs from traditional microservices as the digest is generated immediately after the completion of each invocation. Without FUSE, S1, S2, S3, and S4 have an average latency of 13 milliseconds (ms), 14 ms, 5 ms, and 6 ms, respectively. In contrast, the average latency of S1, S2, S3, and S4 increases to 235 ms, 204 ms, 168 ms, and 276 ms with FUSE. The increased latency comes from processing the traces and the generation and notification of digest, which will be optimised in future work.

5.3 Faults Detected by FUSE

FUSE detected 53 faults at runtime for services S1 and S2 while S3 and S4 did not experience any faults, meaning all the traces for S3 and S4 came from $\mathcal{T}$ listed in Table 1. In contrast, for S1 and S2, FUSE detected faults with random traces that are unfound in $\mathcal{T}$ in Table 1. These 53 faults are because S1 and S2 rely on a MySQL database, which crashed due to being out of memory as *OOM killer* terminated the MySQL process, generating faulty traces. These traces indicated early signs of memory issues as they had additional system calls. The services ran correctly as long as there was enough memory, and then it started to experience faults, resulting in random and unknown traces, triggering faults. For S1, the observed trace during fault includes (8507), while for S2, the faulty traces include (3173, 913b, 51a7). These faults underscore FUSE's effectiveness in capturing runtime faults.

6 Discussion

Tool Selection for Digest Generation: eBPF offers a robust toolkit for generating traces of user-level programs, including microservices. This study has selected specific tools focused on system calls, function invocations, and disk operations, as highlighted in Table 1. The resulting unique digests at the kernel level attest to the effectiveness of these chosen tools. Nevertheless, there remains untapped potential in expanding FUSE's trace generation capabilities to uncover hidden faults and runtime inconsistencies, representing an exciting avenue for future research. **Impact of Inputs on the Digest:** Each service invocation's uniqueness arises from varying input characteristics. Many microservices validate inputs for error prevention prior to processing. Our evaluation has

primarily considered microservices with typical inputs. However, there is room for more in-depth analysis by drastically altering input parameters, offering a promising area for future exploration. **Platform-Agnostic Traces:** This evaluation employed AWS EC2 instances running an Ubuntu operating system to validate FUSE. A valuable opportunity exists to execute the same microservices on diverse eBPF-compatible Linux systems, such as Amazon Linux, to assess the platform-agnostic nature of FUSE's traces. This paper assumes FUSE traces are generated in 'idempotency validation' mode on the production platform. However, testing how digests evolve when introducing a new platform could enhance FUSE's flexibility for platform migration, which we consider for future work.

7 Conclusion

Microservices frequently encounter runtime faults stemming from hardware issues, software bugs, and network disruptions. Detecting these faults is crucial for preempting failures and preventing cascading issues. FUSE, an innovative fault diagnosis and suppression tool built on eBPF, distinguishes microservices as idempotent or non-idempotent based on runtime traces. It dynamically identifies runtime faults by comparing actual traces with expected ones and blocks external requests to other services upon fault detection. FUSE introduces a unique stability scoring mechanism, evaluating microservices based on trace consistency and idempotency. A proof-of-concept implementation using eBPF and Flask, deployed on AWS EC2 instances, validates FUSE's practicality. Performance evaluations involving four representative microservices demonstrate FUSE's capacity to detect 53 runtime faults, albeit with some latency and storage overhead. Future work includes optimizing FUSE's performance through customized eBPF probes, confirming platform agnosticism across various eBPF-compatible Linux platforms, and enhancing its capabilities to analyze input impact on digests by varying inputs significantly in test services.

References

1. Cerveira, F., Oliveira, R.A., Barbosa, R., Madeira, H.: Evaluation of restful frameworks under soft errors. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), pp. 369–379. IEEE (2020)
2. Chandramouli, R.: Microservices-based application systems. NIST Spec. Publ. **800**(204), 800–204 (2019)
3. Chang, H., Kodialam, M., Lakshman, T.V., Mukherjee, S., Van der Merwe, J., Zaheer, Z.: MAGNet: machine learning guided application-aware networking for data centers. IEEE Trans. Cloud Comput. **11**(1), 291–307 (2023)
4. Chang, H., Kodialam, M., Lakshman, T., Mukherjee, S.: Microservice fingerprinting and classification using machine learning. In: 2019 IEEE 27th International Conference on Network Protocols (ICNP), pp. 1–11 (2019)
5. Constantinescu, C.: Intermittent faults and effects on reliability of integrated circuits. In: 2008 Annual Reliability and Maintainability Symposium, pp. 370–374. IEEE (2008)

6. Dixit, H.D., et al.: Silent data corruptions at scale. arXiv preprint arXiv:2102.11245 (2021)
7. Dragoni, N., et al.: Microservices: Yesterday, Today, and Tomorrow, pp. 195–216. Springer, Cham (2017)
8. Fulton III, S.M.: What led amazon to its own microservices architecture. The New Stack (2015)
9. Goldshtein, S.: The Next Linux Superpower: eBPF Primer. USENIX Association, Dublin (2016)
10. Hartono, A.P.P., Fetzer, C.: BROFY: towards essential integrity protection for microservices. In: 2021 40th International Symposium on Reliable Distributed Systems (SRDS), pp. 154–163. IEEE (2021)
11. Jagadeesan, L.J., Mendiratta, V.B.: When failure is (not) an option: reliability models for microservices architectures. In: 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 19–24. IEEE (2020)
12. Kakivaya, G., et al.: Service fabric: a distributed platform for building microservices in the cloud. In: Proceedings of the Thirteenth EuroSys Conference, pp. 1–15 (2018)
13. Levin, J., Benson, T.A.: ViperProbe: rethinking microservice observability with eBPF. In: 2020 IEEE 9th International Conference on Cloud Networking (CloudNet), pp. 1–8 (2020)
14. Li, W., Lemieux, Y., Gao, J., Zhao, Z., Han, Y.: Service mesh: challenges, state of the art, and future research opportunities. In: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 122–1225 (2019)
15. McCanne, S., Jacobson, V.: The BSD packet filter: a new architecture for user-level packet capture. In: USENIX Winter, vol. 46 (1993)
16. Microservices, B.J., Varanasi, B., Bartkov, M.: Spring REST. Springer, Berkeley (2021). https://doi.org/10.1007/978-1-4842-0823-6
17. Power, A., Kotonya, G.: A microservices architecture for reactive and proactive fault tolerance in IoT systems. In: 2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM), pp. 588–599 (2018)
18. Ranjitha, K., Tammana, P., Kannan, P.G., Naik, P.: A case for cross-domain observability to debug performance issues in microservices. In: 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), pp. 244–246. IEEE (2022)
19. Samir, A., Pahl, C.: DLA: detecting and localizing anomalies in containerized microservice architectures using Markov models. In: 2019 7th International Conference on Future Internet of Things and Cloud (FiCloud), pp. 205–213 (2019)
20. Sharma, P., Porras, P., Cheung, S., Carpenter, J., Yegneswaran, V.: Scalable microservice forensics and stability assessment using variational autoencoders (2021)
21. Singleton, A.: The economics of microservices. IEEE Cloud Comput. **3**(5), 16–20 (2016)
22. Weng, T., Yang, W., Yu, G., Chen, P., Cui, J., Zhang, C.: Kmon: an in-kernel transparent monitoring system for microservice systems with eBPF. In: 2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence), pp. 25–30 (2021)