

A System for Identification of Potentially Sensitive Data in the Cloud

Nikola Nemeš

Aalto University
School of Science
Master's Programme in Security and Cloud Computing

Thesis submitted for examination for the degree of Master of
Science in Security and Cloud Computing.

Espoo July 29, 2022

Supervisors

Prof. Raja Appuswamy, EURECOM
Prof. Mario Di Francesco, Aalto University

Advisor

M.Sc. Joshua Burke

Copyright © 2022 Nikola Nemeš





AuthorNikola Nemeš

TitleA System for Identification of Potentially Sensitive Data in the Cloud

School School of Science

Degree programmeMaster's Programme in Security and Cloud Computing

Major Security and Cloud Computing (SECCLO)**Code** SCI3113

Supervisors

Prof. Raja Appuswamy, EURECOM

Prof. Mario Di Francesco, Aalto University

Level Master's thesis**Date** July 29, 2022**Pages** 48**Language** English

Abstract

Due to the increasing amount of digital services that handle sensitive data such as personal information, health information or financial information, data breaches are becoming more common and more costly. The cost of an average data breach in 2021 was USD 4.24 million. DLP solutions present a new and developing security paradigm that focuses on preventing data breaches. Unlike traditional security mechanisms, which analyze metadata and access rights, DLP solutions focus on analyzing content. Depending on the type of data is being analyzed, a wide range of data analysis methods can be used, such as regular expressions, fingerprinting methods or statistical methods. While many DLP solutions offer novel approaches in the dimension of data analysis methods, they do not focus on the usability aspect of defining data protection policies. In this thesis we explore the possibility of a solution that supports data protection policy definition using an interpreted DSL. Our solution aims to provide users with the ability to define data protection policies in an easily readable format that is based on the core concepts of the DLP paradigm. However, the interpretation of the DSL incurs certain performance overhead compared to native execution. Due to this, we make suggestions as to how the solution can be further improved upon, allowing it to reach minimal to no performance overhead, while also providing users with a new approach for defining data protection policies.

Keywords: cloud security, log analysis, data leakage prevention, domain specific languages



AuteurNikola Nemeš

TitreA System for Identification of Potentially Sensitive Data in the Cloud

Programme d' Études Double Diplôme de Master

Filière d'Attachement Security and Cloud Computing (SECCL0)

Encadrants Académiques Prof. Raja Appuswamy, EURECOM
Prof. Mario Di Francesco, Aalto University

Categorie La Thèse de Master **Date** July 29, 2022 **Pages** 48 **Langue** Anglais

Abstrait

On constate que l'utilisation accrue des services numériques qui traitent des données sensibles, notamment les informations personnelles, financières, et sur la santé. Par conséquent, les violations de données deviennent plus fréquentes et plus coûteuses; en fait, le coût moyen d'une violation de données en 2021 a été évalué à 4,24 millions USD. Les solutions DLP présentent un nouveau modèle de sécurité dont le seul objectif est de prévenir les violations de données. Contrairement aux mécanismes de sécurité traditionnels, ces solutions se concentrent sur l'analyse du contenu, et non sur l'analyse des métadonnées et des droits d'accès. Ces solutions proposent d'utiliser plusieurs méthodes d'analyse par type de données analysées, comme les expressions régulières, les méthodes d'empreinte digitale ou les méthodes statistiques. Néanmoins, ils ne se concentrent pas sur l'aspect convivialité de la définition des politiques de protection des données. Dans cette thèse, nous proposons une solution qui permet de définir la politique de protection des données à l'aide d'un DSL interprété. Notre solution offre aux utilisateurs la possibilité de définir des politiques de protection des données dans un format facilement lisible, basé sur les concepts fondamentaux du paradigme DLP. Notre solution offre aux utilisateurs la possibilité de définir des politiques de protection des données dans un format facilement lisible, basé sur les concepts fondamentaux du paradigme DLP. Cependant, l'interprétation du DSL ajoute une dégradation de performances par rapport à l'exécution native. Pour cette raison, nous proposerons comment améliorer encore la solution pour qu'elle atteigne un surcoût de performance minimale ou nul, tout en fournissant une nouvelle approche pour définir les politiques de protection des données.

Mots-clés: cloud security, log analysis, data leakage prevention, domain specific languages

Preface

I would like to thank my university supervisors, **Professor Raja Appuswamy** and **Professor Mario Di Francesco**, for allowing me to research this topic and for giving their input on the direction of the thesis and the thesis itself. I would also like to thank my company advisor, **Joshua Burke**, who introduced me to the topic and also fostered an open company environment which allowed me to freely explore the topic. Furthermore, I would like to thank the SECULO consortium for granting me the amazing opportunity to study at two excellent universities and allowing me to not only build upon my skills as a software engineer but to also explore more of the world and broaden my horizons.

I am also grateful to my family and friends, for their continued support through the thesis writing process and my studies as a whole. I would like to thank my recurring project team members and dear friends, **Aleksandar Nedaković**, **Branislav Anđelić**, **Dušan Milunović** and **Nenad Mišić** for collaborating with me on numerous university projects and for all the fun experiences we shared.

Furthermore, I would like to thank my mother, **Katarina Fedoš Nemeš**, who has always pushed me forward through my studies and supported me in continuing my studies abroad.

Finally, I would like to specially thank my brother, **Ivan Nemeš**, who as a software engineer has served as a role model for me and guided me since the beginning of my studies.

I warrant that the thesis is my original work and that I have not received outside assistance. Only the sources cited have been used in this thesis. Parts that are direct quotes or paraphrases are identified as such

With the support of the
Erasmus+ Programme
of the European Union



Contents

Abstract	2
Abstrait	3
Preface	4
Contents	5
Abbreviations	7
1 Introduction	8
1.1 Motivation	8
1.2 Contribution	9
1.3 Structure of the Thesis	9
2 Background	11
2.1 Virtualization	11
2.1.1 Virtual Machines	11
2.1.2 Containers	12
2.1.3 Container Log Collection	14
2.2 Kubernetes	14
2.2.1 Cluster Architecture	14
2.2.2 Kubernetes Log Collection	16
2.3 Domain Specific Languages	17
2.3.1 DSL Design and Development	18
3 Sensitive Data Detection	21
3.1 Types of Sensitive Data	21
3.2 Data Leakage Prevention Systems	22
3.2.1 Categorization of Data Leakage Prevention Systems	22
3.3 Data Analysis Methods	24
3.3.1 Regular Expressions	24
3.3.2 Fingerprinting	24
3.3.3 Statistical Methods	28
3.3.4 Other data analysis methods	28
3.4 Challenges	29
3.5 Threat Model	30
4 Our Approach to Sensitive Data Detection	31
4.1 The Goal of our Solution	31
4.2 Used Technology Stack and Solution Architecture	31
4.2.1 Log Collection	31
4.2.2 Log Analysis	32
4.3 Our DSL	35
4.3.1 File Selection	35

4.3.2	Scanning Logic	37
4.3.3	Custom or Highly Computational Logic	37
4.3.4	Complete DSL example	39
5	Evaluation	40
5.1	Performance of an Interpreted DSL	40
5.2	Performance of batching operations	43
6	Conclusion	45

Abbreviations

BNF	Backus-Naur form
DLP	Data Leakage Prevention
DLPS	Data Leakage Prevention System
DSL	Domain Specific Language
GDPR	General Data Protection Regulation
GPL	General-purpose Programming Language
IP	Intellectual Property
K8S	Kubernetes
OCI	Open Container Initiative
OS	Operating System
PFI	Personal Financial Information
PHI	Protected Health Information
PII	Personally Identifiable Information
SVM	Support Vector Machine
TF-IDF	Term Frequency-Inverse Document Frequency
VM	Virtual Machine
VMM	Virtual Machine Monitor

1 Introduction

1.1 Motivation

The prime motivation for detecting sensitive data is to avoid potential data breaches and to reduce the overall attack surface of a system. According to [20], a data breach can be defined as any viewing, releasing, stealing or using of sensitive, protected or confidential information by any party which was not authorized to do so. The impact of a data breach depends on the amount and type of leaked sensitive information. Sensitive data can range from critical system data (confidential information, system credentials) to sensitive individual information (personally identifiable information, financial records, medical records).

According to a joint yearly report done by IBM and the Ponemon Institute [7], the average cost of a data breach in 2021 was USD 4.24 million, the highest recorded in the 17 year history of the report. Furthermore, according to the report, 20% of the data breaches were caused by compromised (leaked) credentials. In [33], the researchers developed a tool for identifying Google Play Store applications deployed on mainstream cloud provider servers with data leakage vulnerabilities. Out of a set of 1.6 million applications, 15 098 were found to be vulnerable.

For the purpose of preventing data leakage, many DLPSs (Data Leakage Prevention Systems) are being actively developed. Unlike conventional security mechanisms such as Firewalls and VPNs (Virtual Private Networks) which focus on restricting access to information, DLPSs focus on analyzing the content itself [2] and notifying the data owner. Many widespread industry DLPSs already exist, such as Amazon Macie¹, Cloud Data Loss Prevention by Google² and Azure Named Entity Recognition³.

In the case of the EU, the GDPR⁴ (General Data Protection Regulation) warrants a special mention. The GDPR places additional responsibility on companies handling data of EU citizens in regards to taking proper measures in avoiding data breaches and adhering to a strict protocol of properly reacting to data breaches. While not explicitly required by the GDPR, DLPSs help companies greatly in avoiding GDPR infringements and costly fines.

¹<https://aws.amazon.com/macie/>

²<https://cloud.google.com/dlp/>

³<https://docs.microsoft.com/en-us/azure/cognitive-services/language-service/named-entity-recognition/overview>

⁴<https://gdpr-info.eu/>

1.2 Contribution

This thesis explores detection of potentially sensitive data using a DSL (domain specific language).

The motivation behind leveraging a DSL is twofold:

- The concept of sensitive data is relative and context-sensitive. Different business domains have varying security requirements and business rules that determine the sensitivity of data. For instance, old documents can be declassified or companies can have their own uniquely formatted internal identifiers. A DSL is able to meet these requirements.
- Rules and policies for detection and handling of sensitive data are business-level logic. As such, it is much more natural to express said rules and policies with a DSL rather than a general purpose programming language.

Current solutions tend to have fixed types of sensitive data that they support locating, without a high degree of customizability or context-driven analysis. This solution aims to address these aspects of sensitive data detection.

The contributions of this thesis are the following:

- An overview of the DLPS paradigm that includes:
 - Categorization and definition of sensitive data
 - Categorization and design for DLPSs
 - Challenges relating to DLPSs
 - Threat models for DLPSs
- A DSL for specifying data sensitivity policies
- A DLPS for context-sensitive data analysis based on a DSL

1.3 Structure of the Thesis

The rest of the thesis is structured as follows:

The second chapter contains necessary background information needed for understanding the main topic of the thesis. It contains brief explanations regarding

the cloud-related aspects of the thesis (Virtualization and Kubernetes) and also an introduction to DSLs.

The third chapter pertains to the main topic of the thesis. It contains the definition of varying sensitive data types, the challenges of collecting and analyzing sensitive data in a cloud environment and possible architecture choices and threat modelling for DLPSs.

The fourth chapter relates to our implementation of a DLPS. It goes over the used technology stack and contains justifications of architecture and design choices. Finally, it describes our DSL for specifying data sensitivity policies and lists DSL snippets (examples).

The fifth chapter contains an evaluation of our solution, it explores the effect employing an interpreted DSL has on the performance of a DLPS.

The sixth and final chapter contains the conclusion and summary of the thesis, while also suggesting improvements and possible future work.

2 Background

2.1 Virtualization

Virtualization is the key concept that allows cloud providers to leverage multi-tenancy on individual physical machines, allowing them to share individual bare-metal (physical) machines between multiple customers. While multi-tenancy allows cloud providers to be economically viable (allowing for greater utilization of hardware) it also poses certain security issues [1, 13].

Virtualization can be done on varying abstraction levels. The two main methods of virtualization are *VMs* (virtual machines) and *Containers*. While virtual machines rely on *Hardware Virtualization*, containers rely on *Operating System-level Virtualization*.

Virtualization offers multiple benefits to cloud providers:

- Isolation of processes: Processes on the same host machine are unaware of each other and cannot communicate among themselves. This provides privacy and security guarantees for individual customers.
- Resource management: The host machine can limit the amount of resources a single process can use. Allowing the cloud provider to charge clients based on used computing power and to stop individual client processes from starving other client processes on the same physical machine.
- Packaging and scaling: Both VMs and Containers contain all the dependencies the target client application requires in order to run, meaning it's simple to replicate them multiple times across a single or multiple physical machines. This allows clients to effortlessly scale the amount of cloud provider hardware they utilize based on their current needs.

2.1.1 Virtual Machines

Virtual Machines have historically been an important research area since the late 1960s. Initially, virtual machines were used to enable sharing of a single mainframe computer between multiple users. With the switch towards cheaper commodity hardware they lost their necessity for a brief period but regained it once better utilization of hardware was an issue once again [23].

Virtual Machines achieve virtualization through a thin software layer called the *hypervisor*, also known as a *VMM* (*Virtual Machine Monitor*). The hypervisor

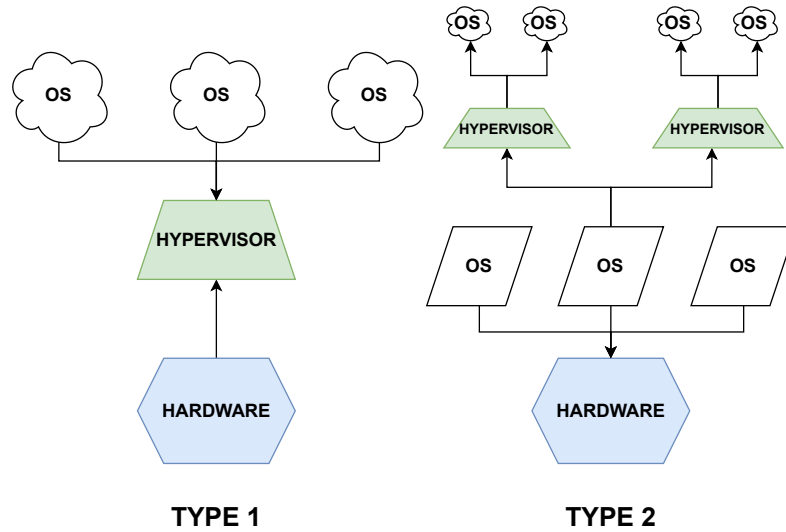


Figure 1: Type 1 (left) and Type 2 (right) VM Hypervisor comparison. The main distinction being that type 1 hypervisors run on hardware, whereas type 2 hypervisors run on top of a host operating system.

abstracts hardware away from the operating system, allowing multiple operating systems to run on the same hardware [9].

There are two types of hypervisors:

- *Type 1 Hypervisor*: A type 1 hypervisor runs directly on the physical machine's hardware and schedules and allocates resources to multiple guest operating systems.
- *Type 2 Hypervisor*: A type 2 hypervisor runs as an application within the host operating system. It is treated as any other process within the host operating system. In this case any system calls made within the guest operating systems are trapped by the hypervisor and subsequently handled by the host operating system.

A visual representation of the difference between type 1 and type 2 hypervisors can be seen on Figure 1. Within cloud environments, type 1 hypervisors are mostly used.

2.1.2 Containers

Containers are seen as a light-weight and agile method of achieving virtualization. Instead of relying on hardware virtualization, containers rely on virtualization of OS resources, with all containers sharing a single kernel. In Linux, this is achieved by utilizing *namespaces* (to provide isolation) and *cgroups* (to provide resource

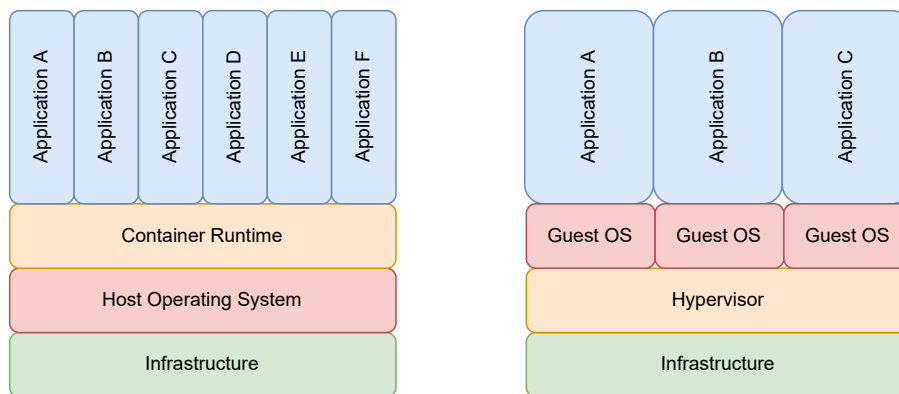


Figure 2: Container (left) and VM (right) comparison. The most notable difference is that each individual VM requires its own guest OS, whereas all containers share a single kernel, allowing for greater density.

management). Containers can break isolation and access the host machines storage in a controlled manner through **bind mounts** [5]. When a bind mount is used, a file or directory from the host machine is mounted within the container filesystem. One notable advantage of this level of virtualization is that each individual container does not require its own individual guest OS, allowing for a larger number of containers to be run on the same machine. While this is advantageous, containers also provide a lower level of isolation compared to VMs [14]. A visual representation of the differences between VM and container virtualization can be seen on Figure 2.

All of the information necessary for running a single container (source code, libraries, dependencies) is specified within a single file called an *Image*. The management of all containers running on a single machine is done by a *Container Runtime*. In order to allow for maximum interoperability between varying image formats and container runtime implementations, the Open Container Initiative (OCI) was formed. All industry image formats and container runtimes follow the OCI Image Format Specification⁵ and OCI Runtime Specification⁶, respectively. OCI also provides the *de facto* industry standard runtime *runc*⁷. The most widely used framework for deploying and managing containers is *Docker*⁸. Docker uses *runc* as its container runtime.

⁵<https://github.com/opencontainers/image-spec>

⁶<https://github.com/opencontainers/runtime-spec>

⁷<https://github.com/opencontainers/runc>

⁸<https://www.docker.com/>

2.1.3 Container Log Collection

While virtualization allows for all of the previously listed benefits (isolation, packaging, scaling and decoupling from hardware), it has an effect on data collection and log analysis. Due to the isolated and ephemeral nature of containers and VMs, logs cannot be collected in the same way as with traditional deployments on bare-metal machines.

In the case of containers, a special log collection mechanism is made available via *logging drivers* [6]. Logging drivers capture a container's *stdout* and *stderr* streams and send the data to a destination designated by the logging driver implementation. By default, docker uses the **json-file** logging driver which saves all container logs within json files on the host machine. Other logging drivers might send the logs to a service (Systemd, Journald) that is locally hosted on the same machine or even over the network to a remote service. While logging drivers suffice for most containerized applications (ones that output their logs to *stdout* and *stderr*), any application that outputs their logs within the container's filesystem instead is not captured by the logging driver. In this case, binding of storage between host OS and container is necessary, or in the case of Kubernetes (where a container can be scheduled on any of multiple machines) the *sidecar-container* pattern is used [19].

2.2 Kubernetes

Kubernetes (K8S) is the *de facto* industry standard platform for container orchestration⁹. It provides a declarative language for stating the desired cluster state and maintains it automatically. On any change (cluster failure or new desired cluster state), it attempts to achieve the desired state again automatically. It achieves this by using the *controller pattern*¹⁰.

2.2.1 Cluster Architecture

On a high level, a Kubernetes cluster is split between *Control Plane* components and *Node* components [18]. A visual depiction of Kubernetes architecture can be seen Figure 3. All of the components depicted on the image are described bellow.

The **Control Plane** contains components that manage the Kubernetes cluster. These components handle global cluster decisions, such as pod scheduling, state management and communication management. While control plane components can

⁹<https://kubernetes.io/>

¹⁰<https://kubernetes.io/docs/concepts/architecture/controller/>

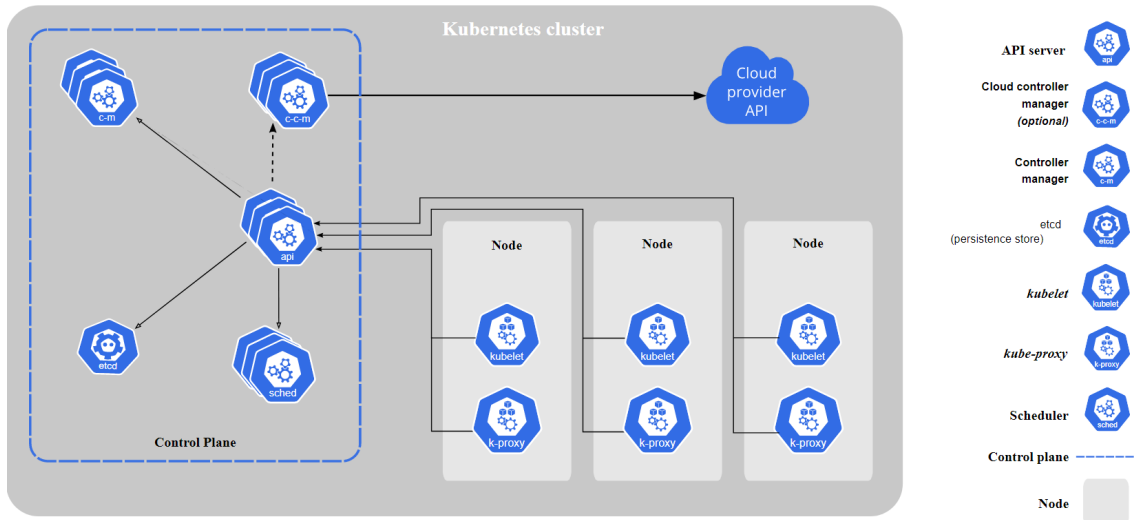


Figure 3: A high-level overview of the Kubernetes cluster architecture. Image taken from [18].

be deployed across numerous machines, for simplicity they can all be deployed within a single machine as well.

Control plane components are:

- The **Kube API server**, a control plane component that exposes the Kubernetes API. It serves as the front end for the Kubernetes control plane.
- The **etcd**, a highly available key value store responsible for storing cluster data.
- The **Kube Scheduler**, a component which assigns newly created pods to worker nodes. It does this scheduling based on resource availability and user configuration.
- The **Kube Controller Manager**, a control plane component that runs all Kubernetes controller processes. As mentioned previously, controllers manage the state of the cluster and automatically attempt to achieve the desired state.
- The **Cloud Controller Manager**, an optional component that embeds cloud-specific control specific to certain cloud providers.

A **node** is either a bare-metal machine or VM worker running within the Kubernetes cluster. Unlike control plane components, the responsibility of node components

is not managing the state of the cluster, but rather running containerized user workloads. A node does not manage containers directly, instead it manages pods. A **pod** is the smallest, most basic unit of deployment within Kubernetes. A single pod can contain one or more containers that are tightly coupled. Containers within the same pod share storage and networking resources.

Each node runs two components:

- The **Kubelet**, an agent that runs on each node within the cluster. The main role of the kubelet is managing pods running on the kubernetes node.
- The **Kube Proxy**, a network proxy that maintains network rules and facilitates communication both from inside and outside the cluster.

2.2.2 Kubernetes Log Collection

As mentioned previously, it is possible to capture and save container logs using logging drivers. However, since there are multiple possible host machines (nodes, which can fail at any time), instead of saving the logs on host machines, log storage is most often centralized by sending all logs to a central repository.

In order to facilitate log collection from individual nodes, logging agents, such as **fluentd**, are required. A logging agent is a piece of software that collects log data on the machine it is deployed on and sends it towards a centralized repository.

There are two common methods of deployment of fluentd on Kubernetes clusters:

- One possibility is to use the docker default **json-file** logging driver for all containers. In this case the logs of all containers running on a single node are stored in a shared directory within the host operating system of the node. Then it is possible to collect these logs using a fluentd container which has access to this shared directory through a bind mount.
- Another possibility is to use fluentd's implementation of the docker logging driver. In this case, the docker daemon sends all logs collected from containers directly to a fluentd host that can be hosted on the same machine or a remote machine.

The previous two methods work assuming that containers send their logs to the *stdout* and *stderr* streams. However, if the logs are saved within individual containers' filesystems, the sidecar container pattern is used. This pattern is Kubernetes specific, since it relies on the tight coupling of containers within a pod, which is a Kubernetes

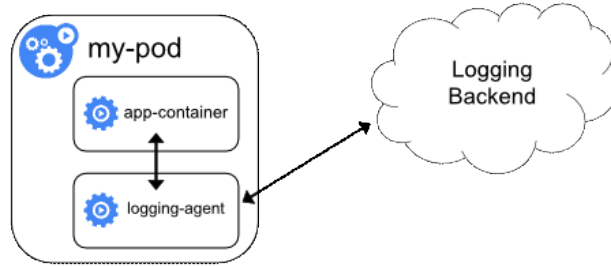


Figure 4: An example of the sidecar container pattern being used. The *logging-agent* (sidecar) container is using the tight coupling provided by the shared pod to access the filesystem of the *app-container* (main container). It streams the analyzed data directly to a logging backend. Image taken from [19].

abstraction. Within this pattern, the behavior of the original container, often called the *main* container is enhanced by the *sidecar* container, an additional container that is deployed within the same pod as the *main* container. When a pod is specified, it is possible to allow multiple containers to share files with each other by mounting the same volume. For log collection purposes, we can mount a volume to the logging directory of the *main* container, whereas the *sidecar* can mount the same volume at any location and access the logs from that location. For instance, a fluentd container can be used as a *sidecar* container, effectively collecting logs from the *main* container without requiring any code changes to the *main* container. A visual depiction of the sidecar container pattern in use can be seen in Figure 4.

2.3 Domain Specific Languages

Domain Specific Languages (DSLs) are formal languages designed for a specific application domain. This is in contrast to *General Purpose Languages (GPLs)* which are designed to be used within a multitude of domains. The motivation behind using DSLs for solving problems is their better expressiveness and usability within a specific application domain compared to a GPL. The justification being that a substantial increase in productivity is gained when using concepts natural to the application domain instead of GPL constructs [17, 4]. Widespread GPLs include languages such as *Python*, *Java*, *C* and *C++*, whereas widespread DSLs include *HTML*, *CSS*, *UML*, and *Latex*. It is not always possible to classify a single language as either a DSL or GPL. For instance, *Cobol*, a language that is still widely used in financial institutions (due to legacy reasons) can be considered as both a GPL and DSL [21].

DSLs can have a wide range of use, such as document description (HTML, Latex, XML), model description (UML) and task description (SQL, Regular Expressions). One major distinction between GPLs and DSLs is that while GPLs are *always* executable, DSLs are not and can just have a descriptive nature (UML). Furthermore, DSLs do not need to even be strictly textual. Use of executable DSLs can provide the same advantage Rule Engines provide, wherein it is possible to change the behavior of an application without changing its source code. Furthermore, since DSLs are also designed to be used by Domain Experts and not software developers exclusively, it is possible to change application behavior without involving software developers.

2.3.1 DSL Design and Development

DSL design and development is a research topic that is not fully formalized. Similarly as there are disagreements whether certain languages are DSLs or GPLs, there are varying attempts of formalization of the DSL design and development process. However, the authors of [21] have written an excellent survey paper where they attempt to better formalize and describe the DSL design and development process.

The authors of the paper have split the development process of a DSL into the following phases: *Decision*, *Analysis*, *Design* and *Implementation*. Each of the phases is independent one from another, meaning any pattern chosen from a single phase can be combined with any pattern from another.

- **Decision:** This phase pertains to making the decision whether to use a DSL to solve a certain problem. Unlike the remaining phases of the development process, this phase answers the question of *'when'* instead of *'how'*. Some possible decision patterns are listed bellow.
 - *Task automation:* Avoiding repetitive tasks in a GPL by introducing a DSL that acts as an *application generator*.
 - *Data structure representation:* Complex data structures can be represented in a more readable and less error-prone way within a DSL in comparison to a GPL.
 - *Data structure traversal:* Traversal of complicated data structures can be done in a more simple way using a DSL in the same manner as data structure representation.
 - *System front-end:* A DSL based System front-end can be used for configuring the behavior of applications. (Instead of GUI configuration)

- **Analysis:** This phase pertains to the analysis of the application domain and extraction of concepts, semantics and processes native to that domain. While formal domain analysis methods do exist, most popular DSLs are formed by informal domain analysis [21].
- **Design:** This phase pertains to patterns of designing the DSL itself. A DSL can be based on existing languages or it can be completely unrelated to any existing language. While basing a DSL on an existing language limits its expressiveness, it reduces the effort needed for implementation and shortens the time needed to learn the DSL.
 - *Piggybacking:* this pattern relies on attaching domain specific features on parts of an existing language. For instance, using algebraic expressions (a well known method of expression) within our DSL.
 - *Restriction:* this pattern relies on limiting the expressiveness of a language to concepts only needed for the specific domain. Restriction and Piggybacking often output similar DSLs, with the main difference being whether the DSL supports features outside of its application domain (Piggybacking) or not (Restriction).
 - *Extension:* this pattern relies on extending an existing language with concepts necessary for solving problems in the application domain.

Furthermore, the complete design of the DSL can be specified in a *formal* or *informal* way.

- *Informal:* usually done through natural language description or visual representation (diagrams).
- *Formal:* done through formal notation such as regular expressions or grammars for syntax specifications (BNF). This method of design specification is especially useful since it can point out inconsistencies within the language design and serve as the core of the implementation of the language.

An important aspect to keep in mind is the usability of the resulting DSL for end-users. When designing a DSL, the intended profile of the end-users should be considered in order to maximize design effectiveness [4]. For instance, if the intended target group of users are users with little to no software development

experience, basing the DSL on an existing GPL does not guarantee good usability.

- **Implementation:** This phase pertains to the selection of an appropriate implementation method for an *executable* DSL.
 - *Interpretation:* Similar to interpreted GPLs, DSL statements are processed in a fetch-decode-execute cycle. This approach is simpler to implement but does not provide optimal performance.
 - *Compilation:* DSL code snippets are compiled into GPL code snippets. These DSLs are also known *application* or *code generators*. This pattern is more difficult to implement compared to interpretation, but it also provides better performance.
 - *Prepossessing:* DSL constructs are translated to GPL constructs during the prepossessing stage when compiling a GPL. In this case DSL and GPL code is mixed before preprocessing.
 - *Embedding:* DSL constructs are embedded within a GPL by defining new abstract types, subroutines and operators. Application libraries and frameworks can be seen as a form of this pattern.
 - *Compiler extension:* A GPL's compiler is extended to support DSL constructs.

3 Sensitive Data Detection

3.1 Types of Sensitive Data

Data leakage is the unauthorized transmission of sensitive data from within an organization to an external destination or recipient [10].

There are varying sensitive data types:

- **PII:** Personally identifiable information - data that can be used to identify a person or their belonging to a certain group.
- **PHI:** Protected health information - health records that identify a person and their health status.
- **PFI:** Personal finance information - financial data of a single person.
- **IP:** Intellectual property - a company's intangible asset that needs to be kept secret for competitive advantage.
- **Confidential Information:** Information that is sensitive and holds commercial value to the company but is not necessarily considered IP.

It's important to note that PII, PHI and PFI relate to information related to individual persons, while IP and Confidential information are sensitive data types that relate to the inner workings of a company or their assets. While leakage of the first three causes damage to the companies' reputation and assets due to fines, leakage of the last two affects the competitive advantage of the company [8]. The most commonly leaked data types can be seen on Table 1. In order to incentivize companies to avoid leakage of personal information (since it causes no loss of competitive advantage), strict regulations and fines are imposed.

Type of Information Leaked	Percentage
Confidential Information	15%
Intellectual Property	4%
Customer Data	73%
Health Records	8%

Table 1: A study of data leakage events categorized by type of leaked data. Data taken from [10]

3.2 Data Leakage Prevention Systems

Data leakage prevention systems are still a relatively fresh security paradigm with no complete categorization and definition. In [25], DLP solutions are defined as products that based on central **policies** identify, monitor and protect *data at rest*, *data in motion* and *data in use* through **deep content analysis**. While more comprehensive definitions exist [2], this one suffices by focusing on the two main aspects of DLP solutions: policy description and deep content analysis. Namely, unlike traditional security mechanisms which provide security through metadata and context analysis, DLP solutions focus on the content itself. However, according to [2], DLP solutions may also employ context analysis (analyzing metadata instead of data) and content tagging (attaching immutable tags to files in order to track them through a system). Furthermore, it is possible to categorize DLP solutions according to varying criteria such as: type of data analyzed [25, 2, 29], deployment method, whether they rely on pattern matching or anomaly detection [8], or data analysis methods. DLP solutions can also have varying threat models, for instance the authors in [26] focus on developing a DLP solution which is resistant to malicious data modification done by insiders which aim attempt to avoid detection by data fingerprinting methods. The authors in [27] focus on developing a privacy enhancing DLP solution in which data analysis can be offloaded to a cloud service provider without allowing the provider to find out exactly which data is sensitive.

3.2.1 Categorization of Data Leakage Prevention Systems

A major categorization of DLP solutions is according to the state of the data they analyze. Possible states of data are: **at rest**, **in motion** and **in use**. The state of the data analyzed also has a big influence on the deployment method of the DLP solution, making these two categorizations intertwined.

- **Data at rest** refers to all data stored within data repositories. This most commonly refers to databases, filesystems and various backups. In this case, the DLP solution focuses on checking access rights, encrypting data at rest or scanning for sensitive data within the data repository. It can be deployed on the same machine as the data repository itself or it can also be deployed on a separate machine and scan the data remotely.
- **Data in motion** refers to data transiting through networks. This can for instance be data that is being sent through emails or various messaging platforms

or files that are being uploaded or downloaded. DLP solutions that protect data in motion are often network based, acting similarly to proxies, analyzing data packets in transit. A notable requirement for Network based DLP solutions is performance. The deployment of such solutions should not noticeably affect network traffic. For this reason, specialized physical DLP appliances that are far more performant than pure software solutions also exist.

- **Data in use** refers to data that is being actively accessed by an user. This can be data that is stored on a user's workstation, such as email attachments, work documents or files accessible through various desktop applications. In this case, the DLP solution is most often deployed as an agent on the workstation, monitoring data that is both arriving to and leaving the workstation.

Another major categorization of DLP solutions is according to the way they attempt to detect data leakage, through pattern matching or anomaly detection.

- **Pattern Matching** solutions analyze and filter data according to predefined patterns. The filtering approach itself can work using whitelisting or blacklisting. In a whitelisting approach, we specify the structure of data that may pass through the system (raising an alert when discovering anything that does not match the pattern), while in a blacklist approach we specify the structure of data that must not appear within the scanned data. The major drawback of these solutions is successfully defining rules that filter data properly and also protect against various newly emerging attacks. A pattern matching approach can only protect against previously known attack vectors.
- **Anomaly Detection** solutions try to prevent data leakage by monitoring the behavior in a system and defining 'normal' or baseline behavior that is allowed, raising an alert when noticing any deviation from normal behavior. While anomaly detection solutions are capable of defending against previously unknown attacks, they have a high rate of false positives, marking any behavior out of the norm as a potential data leakage attack. Anomaly detection solutions are often built upon machine learning models or rule engines which can more easily describe what is expected behavior and what is not.

3.3 Data Analysis Methods

The core of a DLP solution are the data analysis methods it uses. The three main content analysis methods are **regular expressions**, **fingerprinting**, and **statistical methods** [2, 25].

3.3.1 Regular Expressions

Regular Expressions are one of the most common data analysis techniques used within DLP solutions. They allow for fast and effortless matching of well structured sensitive data. Regular expressions are most effective when matching well structured personal information, such as social security numbers and credit card numbers. However, they are ineffective for matching sensitive data that can be modified in such a way to avoid matching the regular expression, such as intellectual property and confidential information. One of their main weaknesses is also a high false positive rate [25]. While they are not an all-encompassing solution, they can be used for first-pass filtering and as a part of more advanced analysis techniques. For instance, certain regular expression matches can indicate parts of text that warrant further analysis. In order to preserve overall system processing rate and not cause bottlenecks when applied to network packets, faster regular expression analysis techniques such as [32] were developed.

3.3.2 Fingerprinting

Fingerprinting allows us to look for leakage of specific data. When utilizing the fingerprint method, a value is hashed and saved. Then, when scanning incoming data for leaks, the incoming data is hashed and the hash is compared to a list of previously saved hashes that correspond to known sensitive data. One issue fingerprinting methods face is data modification, wherein slight modification of the data can change the resulting hash and stop fingerprinting methods from detecting leaks. One additional benefit of fingerprinting compared to regular expressions is that it can often be used for non-textual sensitive data as well.

Fingerprinting can be done at various levels, the most simplistic one being **whole file fingerprinting**, in which a whole file is hashed. This method is vulnerable to any type of data modification since any change (including file metadata) will affect the hash of the whole file.

In order to reduce vulnerability to data modifications, one potential solution is to hash fixed-sized smaller blocks within a file and compile the hashes into a list, making

$$H(S) = S_0 \cdot 2^{n-1} + S_1 \cdot 2^{n-2} + \dots + S_{n-1} \cdot 2^0$$

S1	1	1	1	0	1	0	1	0	1	0
S2	1	1	0	0	1	0	1	0	1	0

$$FP(S1) = (7, 2, 5, 0) \quad FP(S2) = (6, 2, 5, 0)$$

Figure 5: In this case strings S1 and S2 only differ by a single bit flip on the third bit. Fixed block formation enables data leakage detection in this case because only the first block hash is affected, and the rest of the signature is the same.

a new complex fingerprint. Now, when comparison of fingerprints is done, instead of comparing just two individual hash values which are vulnerable to modification, two lists of hashes are compared. An example of this method working successfully is depicted on Figure 5. However, this method introduces the issue of how to mark individual blocks within a file. In case of bit flips (no deletion or addition), this method is effective. However, splitting a file into fixed-sized blocks would still be very vulnerable to even the slightest data addition or removal. For instance, deleting or adding a single byte near the beginning of the file would shift and change the hashes of all blocks after it. An example of what happens to fixed-sized blocks upon data addition near the beginning of a file is depicted on Figure 6. For this reason, a method that is able to split files into similar blocks even after addition or removal is necessary.

A popular approach for splitting a binary file into blocks (also known as *shingling*) is described in [24]. The method works by creating a (comparably) small sliding window of size w and looking at each consecutive w bytes within a file. For each sequence of w bytes a hash value h is calculated. The hash value can then be divided by another pre-selected value c , and the remainder of the operation can be compared to a third pre-selected value t . If the remainder of the division operation matches t , we can declare the byte at which the window starts as the beginning of a new block. The reasoning behind this method is that the selection of borders between chunks is exclusively locally based (to a distance of w), and data modification in one part of the file will not affect the formation of blocks within another part of the file. When comparing two files, the same w , c and t values need to be used. A depiction of a simplified version of this method working is shown on Figure 7. While any hash

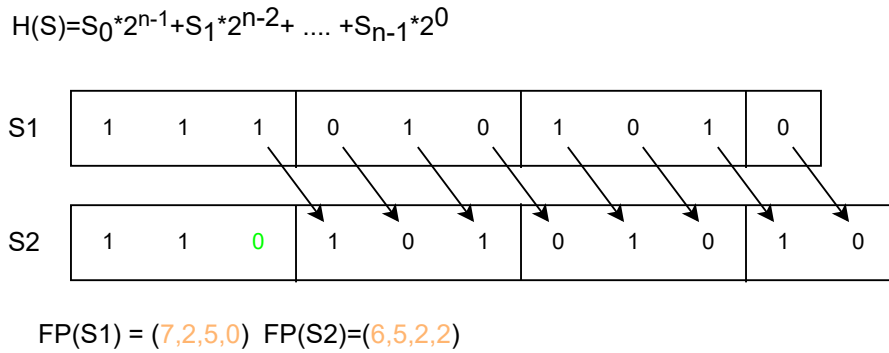


Figure 6: In this case S1 and S2 also differ by only one bit, but instead of bit modification, a new value is inserted in the third position. Now, both the first block and all consequent blocks contain shifted values and their corresponding hash values have changed. In this case data leakage could occur since the resulting fingerprints are completely different.

function could be used to calculate the hash of w bytes, a popular hash function for this approach is the **Rabin fingerprint**. This hashing method is selected because it can be calculated very efficiently in a sliding window scenario. Namely, if we know the hash value for bytes m_0, m_1, \dots, m_w , we can easily calculate the hash value for bytes m_1, m_2, \dots, m_{w+1} by removing the influence of m_0 and adding the influence of m_{w+1} .

In [16], a method similar to the one described above is used. According to the authors, the solution is able to match files that have an overlap as low as only one third of the content (assuming the matching content is at the beginning or at the

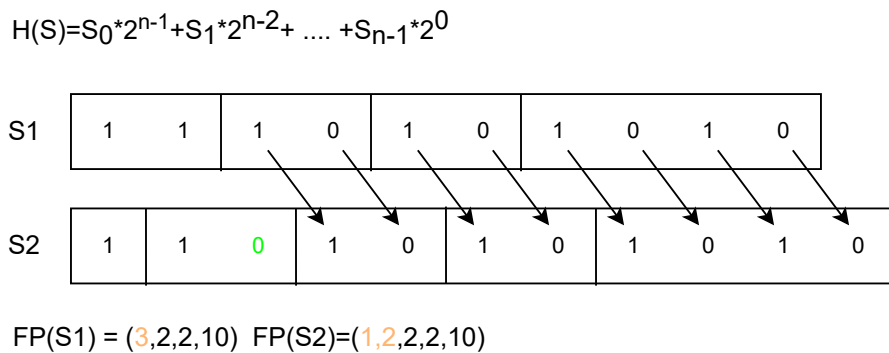


Figure 7: Same string difference as within Figure 6. However, instead of using fixed blocks, we use a simplified version of the algorithm described in [24], with a w of 4. In this simplified version of the method, an anchor point is set before the first bit of the window whenever the window contains the value 1010. Here we can see that the resulting fingerprints have a high degree of similarity, allowing for data leakage detection.

end of the file).

One problem solutions such as [16] don't address is the semantics of fingerprinted data. For instance, two different documents from the same company can have completely different semantics, but share a high percentage of content due to similar document headers or footers. Based on the matching between headers or footers, a false positive data leak can be raised. An example of this is shown in Figure 8. In [26] the authors address this issue by developing a solution that can distinguish between confidential and non-confidential data within documents. The solution requires a training set of both confidential and non-confidential documents clasified by users. During the indexing phase, the algorithm transforms all documents (confidential and non-confidential) into a list of hashes (effectively one fingerprint per file). The individual hashes are based on n-grams of the original document text. Using n-grams as a representation of the document allows for better extraction of semantics within the text. Confidential documents fingerprints are modified by only keeping individual hashes that do not appear often within non-confidential documents. The reasoning behind this is based on **Inverse Document Frequency** [22], meaning

Fingerprinted confidential document	Outgoing document
<p>Irene,</p> <p>What is the status of the pilot timelines? Last week you mentioned that you were waiting for Henry to send you the development timeline for the pilot, and that you were working on communication and planning documents (including timelines) for the pilot. Your assistance in expediting this information would be appreciated.</p> <p>Thank you as always for your cooperation.</p> <p>John Smith T: +111.51.23989132 F: +111.51.23989133 E: john@company.com http://www.company.com</p>	<p>Hi Jack,</p> <p>Do you want to play poker tomorrow night at my place?</p> <p>John Smith T: +111.51.23989132 F: +111.51.23989133 E: john@company.com http://www.company.com</p>

Figure 8: An example of two outgoing emails. The email on the left is confidential while the one on the right is not. Even though the two emails share a high degree of similarity between text, flagging the right email as confidential would be a false positive. Example taken from [26].

that hashes that appear in confidential documents but not in non-confidential ones better characterize the confidential documents. During the detection phase, the fingerprint of the scanned document is calculated and its similarity to fingerprints of the indexing corpus is tested. If a high degree of similarity to confidential documents is found, the scanned document is flagged as confidential.

The problem of detecting leaked confidential data after data modification is closely related to plagiarism detection. For this reason, any existing plagiarism detection methods can also be modified and adapted for use in data leakage prevention.

3.3.3 Statistical Methods

Any statistical or machine learning methods used for text processing can also be applied to DLP solutions. For instance in [11], the authors create a DLP solution based on SVM (support vector machine) machine learning models. While their method achieves high accuracy (a false positive rate of less than 3%), it can only classify documents as confidential or non-confidential, with no possibility of defining varying confidentiality levels. In [3], the authors created a DLP solution that detects sensitive documents based on TF-IDF (Term Frequency-Inverse Document Frequency), a well known statistical analysis method mentioned in this paper as well. This solution provides the capability for the user to define multiple groups (or clusters) of documents according to topic from a training set, with varying levels of confidentiality. If the scanned document gets clustered into a sensitive group, it is marked as a potential data leak.

One issue with statistical methods is their need to be trained on individual enterprise's test data, with labeling being necessary. Finally, as the content of the documents gradually shifts to different topics (for instance when a company changes clients), re-training has to be done.

3.3.4 Other data analysis methods

The previously listed three data analysis methods are most common, but other methods also appear within DLP solutions. For instance, the authors in [28] created a DLP solution that tags files in such a way that the tag is resistant to encryption and data modification. Any file that tries to leave the corporate system containing a tag is recognized as a potential data leak. Furthermore, methods based on rule engines for anomaly detection also exist, a well known example of such a solution is presented in [12].

3.4 Challenges

DLP solutions face numerous challenges. A lot of the challenges that occur within DLP solutions are also related to issues with Big Data solutions [31]. This is because DLP solutions can in many cases deal with big data, due to this, the following challenges apply: (3 Vs of Big Data):

- **Volume:** One challenge for DLP solutions is the sheer amount of data that needs to be analyzed. Solutions need to be written in such a way to be able to handle large amounts of data that cannot be kept in RAM. Furthermore, due to the amount of data, the data may be compressed, making it more difficult to process it efficiently.
- **Variety:** Some DLP solutions face high variety within the data they analyze. While some data analysis methods, such as fingerprinting methods can handle variety in data formats, others cannot. Regular Expressions rely on the data being textual and certain machine learning methods rely on the data being structured.
- **Velocity:** Sometimes DLP solutions need to process and scan data in real-time. For instance, network based DLP solutions have to process the data in such a way that they do not affect network traffic.

However, DLP solutions also face a lot of challenges specific to only them, such as ones mentioned in [15, 2]. These issues can be:

- **Encryption:** Most DLP solutions rely on data analysis methods. However, if the data that reaches the DLP solution is encrypted, it is not possible to use data analysis methods. In this case, only context analysis is possible. An interesting solution for this issue is proposed in [28], wherein files are tagged in such a way that the tagged file is still recognizable even under encryption.
- **Human Factor:** People can act unpredictably and accidentally or intentionally leak shared data. Furthermore, attackers can gain access to sensitive data by circumventing DLP solutions through methods such as social engineering.
- **Leaking Channels:** There is a wide range of channels through which data leakage can occur. For instance, an user can copy the data to a USB drive or a CD, or even simply print it. Even though these channels can be blocked by agent-based DLP solutions, a user can also simply take a picture of their screen or leak the data through an encrypted email or messaging service.

- **Data Modification:** Finally, data modification is also a big issue that DLP solutions have to face. As mentioned previously, there are various data analysis methods that attempt to detect data leakage despite data modification.

3.5 Threat Model

According to [15], the greatest threat for DLP solutions face are insider threats. For this reason, fingerprinting and statistical analysis methods that are resistant to data modification and obfuscation are the primary field of study for DLP solutions. An interesting threat model is proposed in [27]. The authors of this paper propose a DLP solution where the workload of analyzing potentially sensitive data is delegated to a cloud service provider. The DLP solution is designed in such a way to provide privacy-enhancing properties to the owner of the data, by not allowing the cloud service provider to easily deduce which of the data being scanned is actually private. A threat model considering the cloud service provider as a potential threat is often found within privacy enhancing big data related technologies. This type of threat modeling could become more common for DLP solutions, due to the paradigm shift caused by cloud computing, wherein an increasing number of companies choose to adopt cloud services for processing their data.

4 Our Approach to Sensitive Data Detection

4.1 The Goal of our Solution

The goal of our solution is to prevent data leakage within application logs of containers deployed on a Kubernetes cluster. The reasoning of this being that user containers may handle sensitive data and might inadvertently log a sensitive value during operation. Since the company this thesis is being completed in often works with financial institutions, PFI (personal finance information) is our main concern. With PFI being well structured (credit card numbers, bank account numbers), regular expressions must be supported by the solution.

4.2 Used Technology Stack and Solution Architecture

4.2.1 Log Collection

The selected technology stack was partially picked to conform with already used technologies within the company. For this reason, for log collection, **fluentd** was used. A useful capability of fluentd is its capability to format log lines into JSON objects using regular expressions, which is something our solution takes advantage of.

For instance a log line of the following format:

```
1 [Tue Jun 21 21:21:36 2022] [excepturi:notice] [pid 3371:  
   tid 6966] [client 68.241.110.188:1167] You can't  
   calculate the circuit without generating the wireless  
   ADP matrix!
```

Listing 1: Log file snippet

Can be transformed into json object of this format:

```

1 {
2   _id: ObjectId("62ac7b8a9852f10011507c24"),
3   level: 'et:alert',
4   pid: '2985:tid 967',
5   client: '200.90.84.95:13291',
6   message: "I'll back up the back-end SMTP driver, that
7     should array the CSS matrix!",
8   tailed_path: '/var/tmp/logs/host1/log_01.txt',
9   hostname: 'hostname_01_johnny',
10  timestamp: 1655470983,
11  time: ISODate("2022-06-17T13:03:03.000Z")
}
```

Listing 2: Log file snippet

Using this well formatted JSON object, it is easy to extract important parts of information from within the line and use them for our data protection policy.

For central storage of logs MongoDB was selected, as a document database which is meant for storage of JSON objects. This was a convenient choice since fluentd has native support for MongoDB through the *MongoDB output plugin*

An example of our log collection configuration within Kubernetes can be seen on Figure 9. Per Kubernetes node, a single fluentd container collects all logs from containers whose logs are captured by the logging driver. It is able to access the collective logs through a volume bind to the shared docker logging directory on the node. In case of logs that are not captured by the logging driver, a sidecar container with fluentd is used. The sidecar container mounts the logging directory of the main container and reads the logs from there.

An important note regarding the organization of log storage is the naming of collections within MongoDB. Log lines from each individual file are stored within a single collection, which is named based on the hostname of the container and the name of the file. This naming convention of collections within MongoDB allows us to easily reference which logs we need to analyze.

4.2.2 Log Analysis

Components that are responsible for log analysis are split into microservices. Components that have a lot of interaction with the central log repository but little

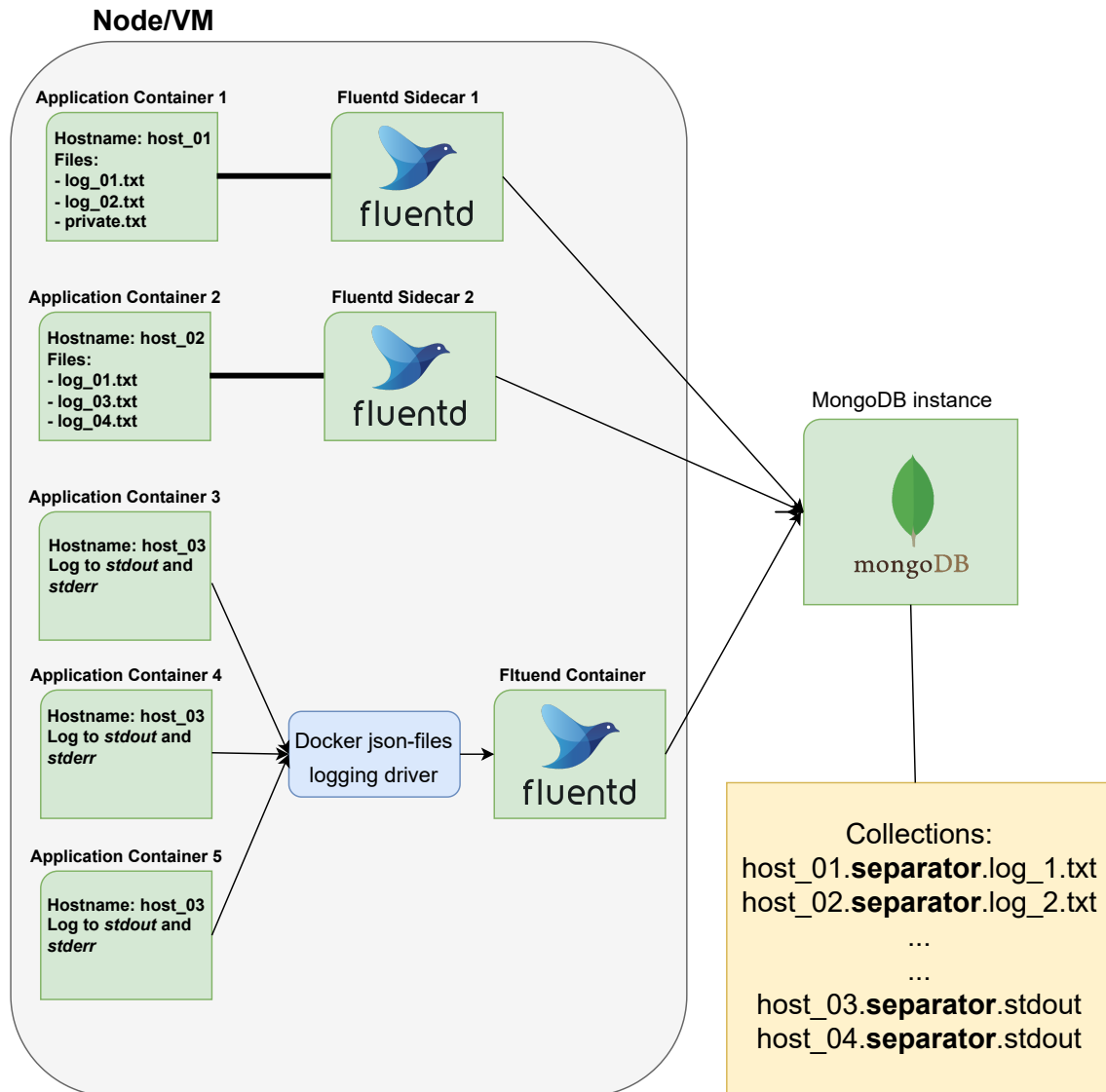


Figure 9: The configuration of the log collection part of our solution.

computationally intensive tasks are implemented in Node.js. This is done because as noted in [30], Node.js provides extremely good performance for services which have a high amount of asynchronous operations (such as retrieving data from a database or waiting for results from an API endpoint). Our solution also supports converting any potentially computationally intensive task within these components to asynchronous tasks, dealing with highly computational workloads as well. All log analysis components are depicted on Figure 10.

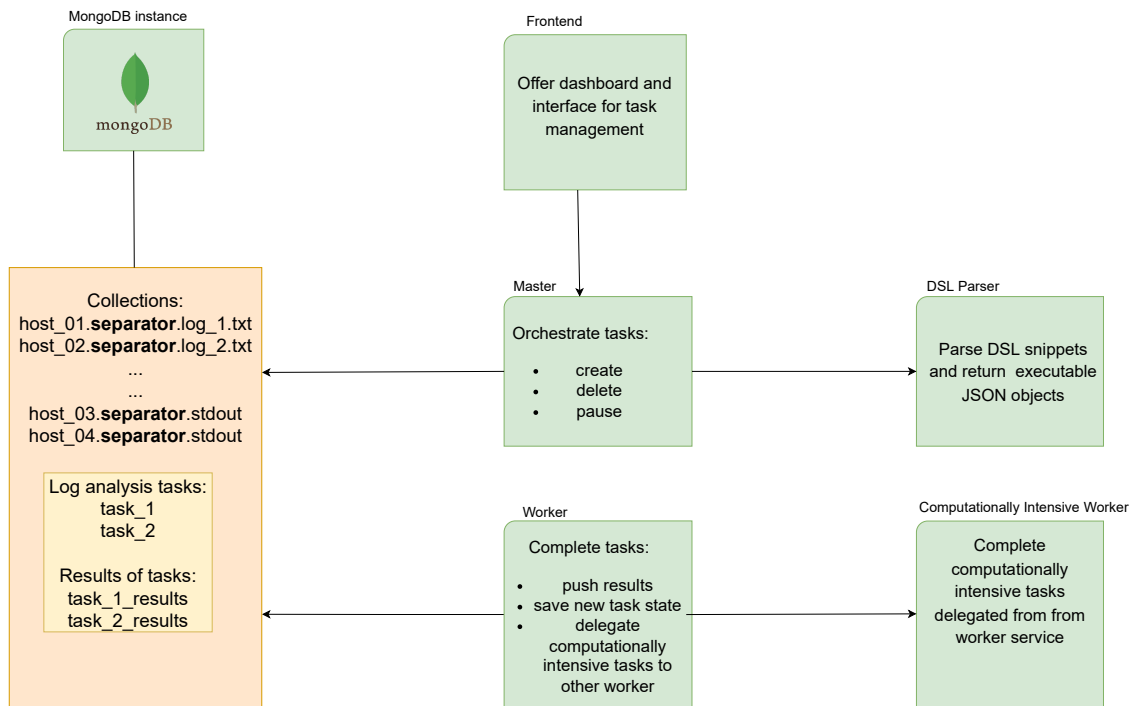


Figure 10: Architecture of the log analysis part of our solution.

The components have the following roles:

- The **Frontend** service is a React based frontend application that allows the user to interact with our DLP solution. It allows the user to view results and manage tasks.
- The **Master** service is a Node.js application that handles task orchestration. Upon requests from the frontend service, it manages the tasks within the database and retrieves task results. When creating tasks it contacts the DSL Parser service to transform DSL snippets into executable tasks that are formatted as JSON objects.
- The **DSL Parser** service is a simple single endpoint service that parses DSL text snippets and returns executable JSON trees.
- The **Worker** service is a Node.js application that queries the database for active tasks and executes them according to the executable JSON trees attached to them. Upon finding any data that can be considered sensitive, it saves the result to the database. If the DSL specifies usage of any computationally intensive tasks, it batches these operations and offloads them to the computationally intensive worker.

- The **Computationally Intensive Worker** is a simple single endpoint service that accepts batched computationally intensive jobs from the worker service and the returns results back to the worker.

4.3 Our DSL

As all domain specific languages, our DSL was defined with the intention of being easy to use for target users. We can quickly go over the DSL design and development patterns mentioned within Chapter 2.3.1 and categorize our DSL according to these specifications.

- **Decision:** A DSL was selected for enhanced readability but also for task automation. Although data protection policies can be described in a GPL, a DSL allows for them to be expressed in a simpler way.
- **Analysis:** As is the case with most DSLs, informal domain analysis was done and concepts which we considered core to the DLP domain and our use case were extracted.
- **Design:** Our language uses the piggybacking pattern, in other words it reuses known languages. Specifically, parts of our DSL are based on boolean expressions and regular expressions. The design of the language is formal, since it is based on a BNF grammar.
- **Implementation:** The language is interpreted. A DSL snippet is transformed into a tree of execution represented in JSON format. The JSON format was selected due to our microservice architecture and due to the need for interoperability across varying GPLs that could potentially execute the DSL.

A snippet of the DSL code is organized in two main sections, the **File Selection** section and the **Scanning Logic** section. Both sections act according to a filtering logic, accepting a boolean expression which determines whether an object progresses to the following stage. In the case of the file section the objects are files and in the case of the scanning logic section the objects are individual log lines.

4.3.1 File Selection

File selection is possible based on two variables, the filename and the hostname of the machine the file is located on. For instance, if we look at Figure 9, all files within

containers can be uniquely identified by a combination of filename and the hostname of the container. In our example different containers contain files with same filenames because this is often true in real-world scenarios. For instance, in clusters it is often the case that the same image is deployed on multiple nodes for higher availability and performance. Below we can see some examples of file selection based on the cluster setup depicted in Figure 9.

```
1 FILE_SELECTION: {  
2     $hostname CONTAINS /host_01/  
3 }
```

Listing 3: Selecting all files from a single container

```
1 FILE_SELECTION: {  
2     $hostname CONTAINS /host_01/  
3     AND  
4     $filename DOES NOT CONTAIN /private/  
5 }
```

Listing 4: Selecting all files from a single container except for private files

```
1 FILE_SELECTION: {  
2     $filename CONTAINS /stdout/  
3 }
```

Listing 5: Match all logs that are outputted to stdout from any container within the cluster

```
1 FILE_SELECTION: {  
2     $filename CONTAINS /stdout/  
3     OR  
4     $hostname CONTAINS /host_02/  
5 }
```

Listing 6: Match all logs that are outputted to stdout and any log from host_02

```

1 FILE_SELECTION: {
2     true
3 }

```

Listing 7: Match all logs within the cluster (regardless of hostname or filename)

4.3.2 Scanning Logic

Scanning logic is based on a boolean expression which uses any attribute extracted from a log line, such as the JSON object depicted in Listing 2. For instance, based on the structure of the previously mentioned JSON object we could write scanning logic that retrieves all *info* or *warning* level log messages that contain the keyword **HDD** in the text with the following DSL snippet:

```

1 SCANNING_LOGIC: {
2     ($level CONTAINS /info/ OR $level CONTAINS /warn/)
3     AND
4     $message CONTAINS /HDD/
5 }

```

Listing 8: Match all info or warning level log lines that reference 'HDD'

Another interesting use case of the scanning logic would be to capture all requests that come from a client from outside of a certain subnet. This could be accomplished by a scanning logic such as:

```

1 SCANNING_LOGIC: {
2     $client DOES NOT CONTAIN /^200\.90\.84/
3 }

```

Listing 9: Match all requests coming from outside the subnet 200.90.84.0/24

4.3.3 Custom or Highly Computational Logic

The only data analysis method our solution currently supports are Regular Expressions. For this reason, we have added a generic batching method to the DSL. In other words, it is possible to define a new operation for the DSL that internally translates into a remote API call. Since doing a remote call incurs performance

overhead and relies on network traffic as well, we attempt to offset this overhead by supporting batching of these operations. This choice is also made because of the use of Node.js as the framework for our worker service. Namely, Node.js handles asynchronous operations efficiently but has issues with computationally intensive tasks since it is by nature single threaded. Implementing highly computational tasks as remote API endpoint calls avoids one of the pitfalls of Node.js while exploiting one of its biggest strengths. Even though the DSL is written as if each log line calls the API endpoint separately, internally calls to any API endpoint are batched and sent once a certain number of requests is queued. To be able to use remote batched operations, first the operation needs to be defined within the DSL code.

An example of an API definition can be seen bellow:

```

1 API_DEFINITIONS: {
2   FINGERPRINT: {
3     URL: "http://api-endpoint:3000/batch-method"
4     BATCH_SIZE: 25
5   }
6 }

```

Listing 10: Example of an API definition within our DSL. A batching method called 'FINGERPRINT' is created which is translated to an API remote call which is batched in such a way to only send requests once 25 parameters are gathered.

Now that an API definition is created, it can be used within the Scanning Logic section on any variable from the JSON object and combined with any other API call or regular expression. For instance, regular expressions and API calls can be combined like in the following snippet:

```

1 SCANNING_LOGIC {
2   ($level CONTAINS /info/ OR $level CONTAINS /warn/)
3   AND
4   FINGERPRINT($message)
5 }

```

Listing 11: A remote API method defined in the previous snippet is used within the Scanning Logic section.

4.3.4 Complete DSL example

Now that each individual section of our DSL design was explained, we can present a complete data protection policy within a single listing.

```
1 NAME: a_data_protection_policy
2
3 API_DEFINITIONS :{
4   FINGERPRINT: {
5     URL: "http://api-endpoint:3000/batch-method"
6     BATCH_SIZE: 25
7   }
8 }
9
10 FILE_SELECTION: { true }
11
12 SCANNING_LOGIC: {
13   FINGERPRINT($message) AND $level CONTAINS /^q.*warn/
14 }
```

Listing 12: An example of a complete data protection policy defined within our DSL.

5 Evaluation

The main focus of evaluating our solution is the performance overhead of using an interpreted DSL, as opposed to a compiled one. Another important issue is the effectiveness of delegating operations from one worker to another, and the performance overhead the additional network traffic incurs. While testing both points, a test database of approximately 130 000 objects was used. Each experiment was repeated five times and the execution time was averaged between these times to accommodate for variance due to interference from other processes running on the operating system while conducting the experiments.

5.1 Performance of an Interpreted DSL

When testing the performance of the execution our DSL, we conducted experiments with varying expression size. The reasoning behind this is because the size of an expression affects the size of the resulting executable JSON tree, which in turn affects execution time. Expressions with a number of operations ranging from 1 to 10 operations were evaluated. Our DSL was compared with both inlined (plainly written code) and evaluated (eval function) Javascript.

In our first set of experiments, we used the AND boolean operator:

```

1 SCANNING_LOGIC: { $message CONTAINS /HDD/ }
2 ...
3 SCANNING_LOGIC: { $message CONTAINS /HDD/ AND ... (8 more
   ) ... AND $message CONTAINS /HDD/}
```

Listing 13: Expression used in first set experiments for our DSL

```

1 log.message.match(new RegExp("HDD"))
2 ...
3 log.message.match(new RegExp("HDD")) && ... (8 more) ...
   && log.message.match(new RegExp("HDD"))
```

Listing 14: Expression used in first set experiments for inlined Javascript

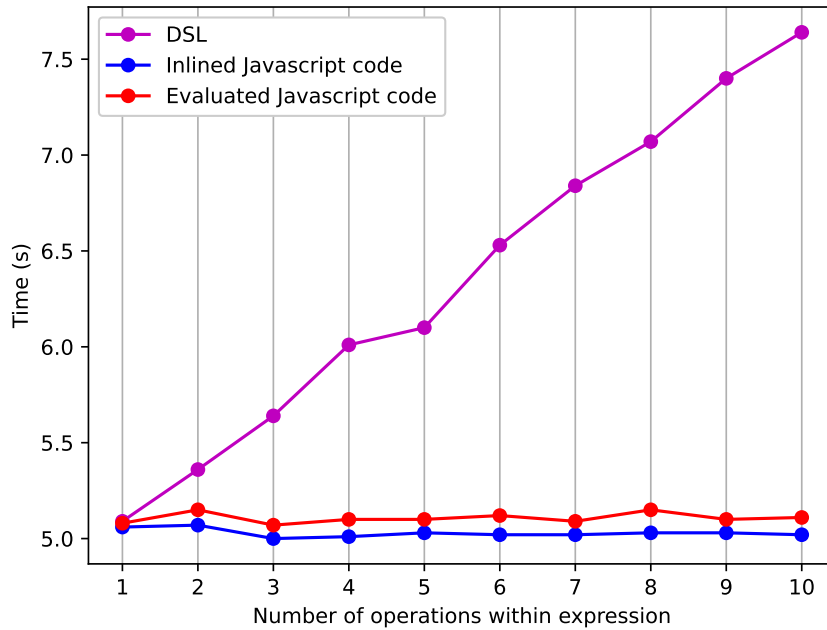


Figure 11: Results of the first set of experiments

```

1 eval('log.message.match(new RegExp("HDD"))')
2 ...
3 eval('log.message.match(new RegExp("HDD")) && ... (8 more
   ) ... && log.message.match(new RegExp("HDD"))')

```

Listing 15: Expression used in first set experiments for evaluated Javascript

The results of running the experiments can be seen in Figure 11. We can see that when the number of operations is low (1-2), the performance overhead of our DSL is negligible, but it rises to a performance overhead of over 50% on an expression containing 10 operations. Evaluated Javascript code is slightly slower than inlined code (due to repetitive calling of the eval function), but both non-DSL implementations have a near constant execution time. A potential explanation for the near constant execution time for the Javascript variants despite increasing expression size can be boolean short-circuit evaluation. Since Javascript natively supports short-circuit evaluation (as almost all GPLs do), the increasing expression size does not affect the performance of the Javascript variants. For this reason, we have conducted the same experiments again for the Javascript variants but with swapping out the AND operations with OR operations (which cannot be short-circuited when

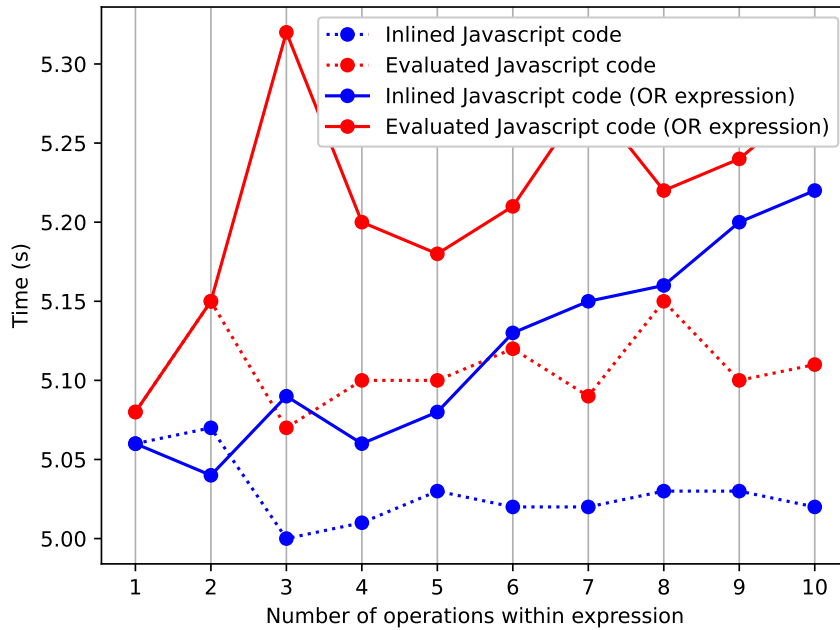


Figure 12: Results of the second set of experiments

each operation evaluates to false often).

The results of these experiments are depicted in Figure 12. Here we can see a near constant increase of execution time with the increase in number of operations within the expression, since no boolean short-circuiting is occurring. However, the increase is not as significant as with our interpreted DSL solution.

From these results we can see that our solution would benefit from either being transformed into a compiled language (Javascript code that can be evaluated) or from implementation of boolean short-circuiting. Even though it does not appear as if boolean-short circuiting is providing big benefits to the Javascript variants, it could provide big benefits for our DSL, since its performance is close to that of the performance of Javascript variants on a small number of operations, as seen in Figure 11. Boolean short-circuiting would allow the execution time to stay at the same length as the execution time when the expression is smaller. Finally, there is also an argument on the need of optimizing the solution further, since a boolean expression containing 10 operations would be quite complex and hard to find in realistic scenarios. Our more 'realistic' examples given in chapter 4.3.2 use a maximum of 3 operations (where the performance of our DSL can be considered acceptable).

5.2 Performance of batching operations

Finally, we compared the performance overhead of our batching API to the execution time of computing the same expression locally within the Node.js worker. Since the Node.js worker only supports Regular Expressions for now, we simply implemented a remote API endpoint that computes a Regular Expression the same way our DSL would.

```
1 FILE_SELECTION: { $message CONTAINS /HDD/ }
```

Listing 16: Expression in DSL that batching is compared against

```
1 request.payload.forEach(message => message.includes("HDD  
  "))
```

Listing 17: Batching Equivalent of same operation

The results of running the code snippets above can be seen in Figure 13. Here we can see that with a high enough batch size, the batched operation nears the performance of the local DSL operation. It should also be noted that this experiment

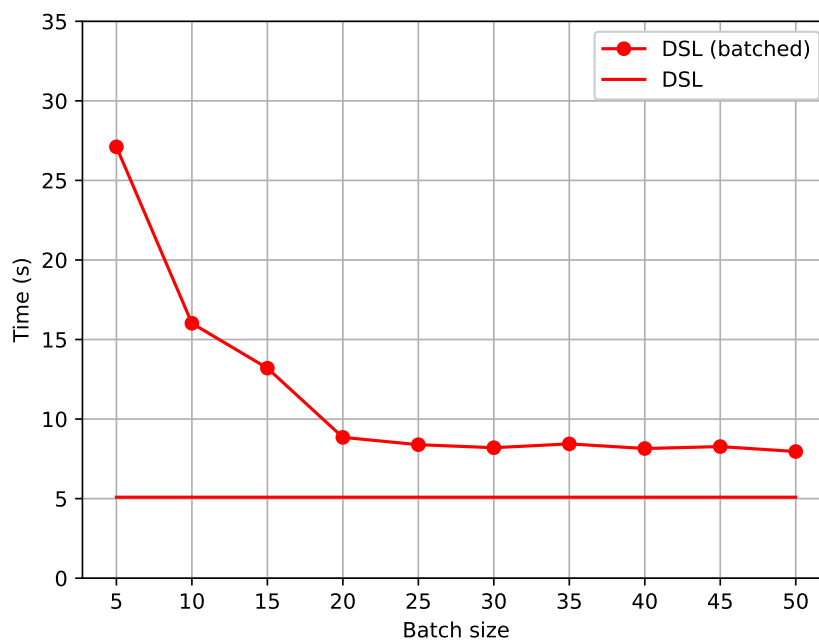


Figure 13: Results of the third set of experiments

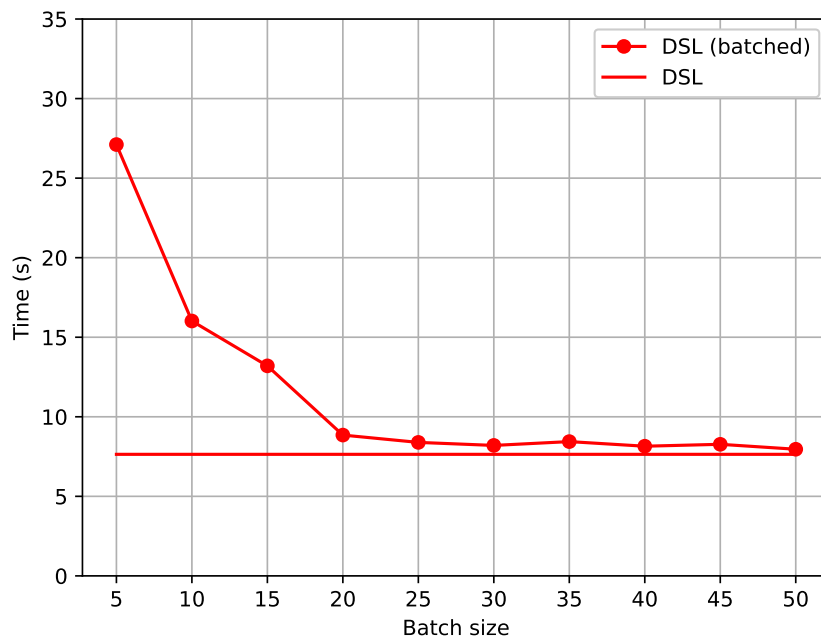


Figure 14: Results of the fourth set of experiments

is meant to highlight the performance overhead of the remote endpoint. Namely, if we pick a more complex expression for both the local DSL and the remote API call, such as one containing 10 operations within a boolean expression, we can see the remote endpoint coming even closer (or even surpassing for complex enough expressions) to local DSL execution. This case can be seen in Figure 14.

6 Conclusion

This thesis explores the DLP paradigm and covers the core concepts related to it. It goes over use cases for varying data analysis methods, varying threat models DLP solutions face and different challenges within DLP solutions. Furthermore it provides a DSL for data protection policy definition and a DLP solution that executes this policy. The DSL provides benefits such as compact and readable data protection policy description that is intuitive to software developers (due to usage of known concepts such as boolean expression and regular expressions). Due to the batched remote API capabilities of the DSL, it provides users an easy way to plug their own data analysis into our DLP solution, having only to write code that marks whether a piece of data is sensitive or not, with our solution providing support for everything else (task management, file selection, log collection). Due to the microservice nature of our solution, the custom API endpoint could be implemented in any technology. From the results of the conducted experiments we can see that although the DSL has acceptable performance, interpretation of the DSL brings some overhead. Based on the results of the experiments, suggestions for improving performance are given (compilation of the DSL or boolean short-circuiting). Although the solution is a good proof of concept for a DLP solution, it could be extended in multiple ways. For instance, an alerting section could be added to the DSL, allowing the user to specify what kind of action can be taken on data leakage detection. Furthermore, a rule engine could be added to the solution to provide support for anomaly detection (which is prominent in DLP solutions) as well. Finally, further improvements to the frontend service could be made (for instance, by adding a Kibana dashboard) to give the user a better overview on the status of the system.

References

- [1] Hussain AlJahdali et al. “Multi-tenancy in cloud computing”. In: *2014 IEEE 8th international symposium on service oriented system engineering*. IEEE. 2014, pp. 344–351.
- [2] Sultan Alneyadi, Elankayer Sithirasenan, and Vallipuram Muthukkumarasamy. “A survey on data leakage prevention systems”. In: *Journal of Network and Computer Applications* 62 (2016), pp. 137–152.
- [3] Sultan Alneyadi, Elankayer Sithirasenan, and Vallipuram Muthukkumarasamy. “Detecting data semantic: a data leakage prevention approach”. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. IEEE. 2015, pp. 910–917.
- [4] Ankica Bariic, Vasco Amaral, and Miguel Goulão. “Usability Evaluation of Domain-Specific Languages”. In: *2012 Eighth International Conference on the Quality of Information and Communications Technology*. 2012, pp. 342–347. DOI: [10.1109/QUATIC.2012.63](https://doi.org/10.1109/QUATIC.2012.63).
- [5] *Bind Mounts*. Accessed: 2020-6-15. URL: <https://docs.docker.com/storage/bind-mounts/>.
- [6] *Configure Logging Drivers*. Accessed: 2020-6-16. URL: <https://docs.docker.com/config/containers/logging/configure/>.
- [7] “Cost of a Data Breach Report 2021”. In: (2021). URL: <https://www.ibm.com/security/data-breach>.
- [8] Elisa Costante et al. “A hybrid framework for data loss prevention and detection”. In: *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2016, pp. 324–333.
- [9] Ankita Desai et al. “Hypervisor: A survey on concepts and taxonomy”. In: *International Journal of Innovative Technology and Exploring Engineering* 2.3 (2013), pp. 222–225.
- [10] Peter Gordon. “Data leakage-threats and mitigation”. In: *InfoSec Reading Room* (2007).
- [11] Michael Hart, Pratyusa Manadhata, and Rob Johnson. “Text classification for data loss prevention”. In: *International Symposium on Privacy Enhancing Technologies Symposium*. Springer. 2011, pp. 18–37.

- [12] Koral Ilgun. “USTAT: A real-time intrusion detection system for UNIX”. In: *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE. 1993, pp. 16–28.
- [13] Amarnath Jasti et al. “Security in multi-tenancy cloud”. In: *44th Annual 2010 IEEE International Carnahan Conference on Security Technology*. IEEE. 2010, pp. 35–41.
- [14] Ann Mary Joy. “Performance comparison between linux containers and virtual machines”. In: *2015 international conference on advances in computer engineering and applications*. IEEE. 2015, pp. 342–346.
- [15] Kamaljeet Kaur, Ishu Gupta, Ashutosh Kumar Singh, et al. “A comparative evaluation of data leakage/loss prevention systems (DLPS)”. In: *Proc. 4th Int. Conf. Computer Science & Information Technology (CS & IT-CSCP)*. 2017, pp. 87–95.
- [16] Jesse Kornblum. “Identifying almost identical files using context triggered piecewise hashing”. In: *Digital investigation 3* (2006), pp. 91–97.
- [17] Tomaž Kosar et al. “A preliminary study on various implementation approaches of domain-specific language”. In: *Information and software technology 50.5* (2008), pp. 390–405.
- [18] *Kubernetes Components*. Accessed: 2020-6-20. URL: <https://kubernetes.io/docs/concepts/overview/components/>.
- [19] *Kubernetes Logging Architecture*. Accessed: 2020-6-20. URL: <https://kubernetes.io/docs/concepts/cluster-administration/logging/>.
- [20] Rakesh Kumar and Rinkaj Goyal. “On cloud security requirements, threats, vulnerabilities and countermeasures: A survey”. In: *Computer Science Review 33* (2019), pp. 1–48.
- [21] Marjan Mernik, Jan Heering, and Anthony M Sloane. “When and how to develop domain-specific languages”. In: *ACM computing surveys (CSUR) 37.4* (2005), pp. 316–344.
- [22] Stephen Robertson. “Understanding inverse document frequency: on theoretical arguments for IDF”. In: *Journal of documentation* (2004).
- [23] M. Rosenblum and T. Garfinkel. “Virtual machine monitors: current technology and future trends”. In: *Computer 38.5* (2005), pp. 39–47. DOI: [10.1109/MC.2005.176](https://doi.org/10.1109/MC.2005.176).

- [24] Vassil Roussev. “Hashing and data fingerprinting in digital forensics”. In: *IEEE Security & Privacy* 7.2 (2009), pp. 49–55.
- [25] LLC Securosis. “Understanding and selecting a data loss prevention solution”. In: *Securosis, LLC* (2010).
- [26] Yuri Shapira, Bracha Shapira, and Asaf Shabtai. “Content-based data leakage detection using extended fingerprinting”. In: *arXiv preprint arXiv:1302.2028* (2013).
- [27] Xiaokui Shu and Danfeng Daphne Yao. “Data leak detection as a service”. In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2012, pp. 222–240.
- [28] Chun Hui Suen et al. “S2logger: End-to-end data tracking mechanism for cloud data provenance”. In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE. 2013, pp. 594–602.
- [29] Radwan Tahboub and Yousef Saleh. “Data Leakage/Loss Prevention Systems (DLP)”. In: *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*. 2014, pp. 1–6. DOI: [10.1109/WCCAIS.2014.6916624](https://doi.org/10.1109/WCCAIS.2014.6916624).
- [30] Stefan Tilkov and Steve Vinoski. “Node.js: Using JavaScript to Build High-Performance Network Programs”. In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83. DOI: [10.1109/MIC.2010.145](https://doi.org/10.1109/MIC.2010.145).
- [31] Alexandru Adrian Tole et al. “Big data challenges”. In: *Database systems journal* 4.3 (2013), pp. 31–40.
- [32] Fang Yu et al. “Fast and memory-efficient regular expression matching for deep packet inspection”. In: *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. 2006, pp. 93–102.
- [33] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. “Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1296–1310. DOI: [10.1109/SP.2019.00009](https://doi.org/10.1109/SP.2019.00009).