NIST Internal Report
NIST IR 8505 ipd

A Data Protection Approach for Cloud-Native Applications

Initial Public Draft

Ramaswamy Chandramouli
Wesley Hales

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8505.ipd

NATIONAL INSTITUTE OF
STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Internal Report
NIST IR 8505 ipd

A Data Protection Approach for Cloud-Native Applications

Initial Public Draft

Ramaswamy Chandramouli
Computer Security Division
Information Technology Laboratory

Wesley Hales
Leak Signal Inc.

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8505.ipd

June 2024

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at https://csrc.nist.gov/publications.

NIST Technical Series Policies

Publication History
Approved by the NIST Editorial Review Board on YYYY-MM-DD [Will be added to final publication]

How to Cite this NIST Technical Series Publication
Chandramouli R, Hales W (2024) A Data Protection Approach for Cloud-Native Applications. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) NIST IR 8505 ipd. https://doi.org/10.6028/NIST.IR.8505.ipd

Author ORCID iDs
Ramaswamy Chandramouli: 0000-0002-7387-5858

Public Comment Period
June 14, 2024 – August 1, 2024

Submit Comments
nistir-8505-comments@nist.gov

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930

All comments are subject to release under the Freedom of Information Act (FOIA).

Abstract

This document addresses the need for effective data protection strategies in the evolving realm of cloud-native network architectures, including multi-cloud environments, service mesh networks, and hybrid infrastructures. By extending foundational data categorization concepts, it provides a framework for aligning data protection approaches with the unknowns of data in transit. Specifically, it explores service mesh architecture, leveraging and emphasizing the capabilities of WebAssembly (WASM) in ensuring robust data protection as sensitive data is transmitted through east-west and north-south communication paths.

Keywords

data governance; data privacy; data protection; data security; in-transit data categorization; WASM.

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation’s measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL’s responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems.

Call for Patent Claims

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the guidance or requirements in this Information Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

a) assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or

b) assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:

    i. under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or

    ii. without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed to: nistir-8505-comments@nist.gov

Table of Contents

1. Introduction

In the constantly evolving landscape of cloud-native application architectures, where data resides in multiple locations (i.e., on-premises and on the cloud), ensuring data security involves more than simply specifying and granting authorization during service requests. It also involves a comprehensive strategy to categorize and analyze data access and leakage as data travels across various protocols (e.g., gRPC, REST-based), especially within ephemeral and scalable microservices applications. As organizations find themselves governing hundreds to tens of thousands of services and the inter-service calls between them, a security void has been identified in observing and protecting sensitive data in transit.

1.1. Existing Approaches to Data Protection and Their Limitations

Traditionally, regular expressions (regex) have been widely used for data categorization to identify patterns that match predefined categories or data classes with the aid of keywords and validators for enhanced precision. Despite its wide adoption and usage, the approach has notable limitations. The processing time scales linearly with data volume, making it impractical for very large datasets. Regex also lacks the capability for logical computations, which are necessary for complex validations like checksums in credit card numbers. Its effectiveness heavily relies on the correct proximity to specific keywords, leading to potential false positives and considerable noise if not managed correctly.

Machine learning (ML) offers a promising enhancement to data categorization by learning from data patterns and improving over time, thus providing a scalable and adaptable solution. ML algorithms can handle both structured and unstructured data, predict data categories based on historical data, and adjust to new patterns without explicit reprogramming. This adaptability significantly reduces the time and computational resources required to manage complex datasets and is effective for both data at rest and in motion.

To address and complement the limitations of traditional data-at-rest inventory, in-transit data categorization has recently come to light as the next logical step in data protection. Unlike the former, which only secures stored information, in-transit categorization actively monitors and secures data as it moves across services and network protocols. This shift to real-time data analysis within the network brings new observability capabilities, eliminating the need for traffic mirroring and data duplication.

1.2. In-Proxy Application for Data Protection

To address the need for data categorization during travel across services, a relatively new class of in-proxy application called the WebAssembly program (also called a WASM module) has been increasingly deployed. A WASM module is a lightweight executable compiled to low-level bytecode. This bytecode can be:

(a) Generated from code written in any language using their associated WebAssembly compilers, including C, C++, and Rust

(b) Run using a WASM runtime in an isolated virtual machine (VM) within the proxy, which allows developers to enhance applications with necessary functionality and run them as efficiently as native code in the proxies.

Over the last few years, the Envoy WASM VM has enabled new types of compute and traffic processing capabilities and allowed for custom WASM modules to be built and deployed in a sandboxed and fault-tolerant manner.

Additionally, the following features of WebAssembly modules make them particularly effective for data protection:

- **Data Discovery and Categorization:** WASM modules can dynamically identify and categorize data as it traverses the network, ensuring that sensitive information is recognized and handled appropriately.
- **Dynamic Data Masking (DDM):** WASM modules can apply DDM techniques to redact or mask sensitive information in transit, enhancing privacy and security.
- **User and Entity Behavior Analytics (UEBA):** WASM modules can analyze user and entity behaviors in real time, detecting anomalies and potential security threats.
- **Data Loss Prevention (DLP):** WASM modules can enforce DLP policies by monitoring and controlling data transfers to prevent unauthorized data exfiltration.

1.3. Objective and Scope of This Document

All services (e.g., networking, security, monitoring, etc.) for microservices-based applications are provided by a centralized infrastructure called the service mesh, and the data plane for this service mesh — which performs all runtime tasks — consists of proxies. This document outlines a practical framework for effective data protection and highlights the versatile capabilities of WebAssembly (WASM) within service mesh architectures, multi-cloud environments, and hybrid (i.e., a combination of on-premises and cloud-based) infrastructures. By focusing on in-line, network traffic analysis at layers 4–7, organizations can enhance security, streamline operations, and utilize adaptive data protection measures.

1.4. Organization of This Document

This document is organized as follows:

- Section 2 describes the execution environment for WASM modules in detail, including the application infrastructure (i.e., service mesh) under which it runs, the specific host environment (i.e., proxies), the process for generating bytecodes and executables, the processes for executing the modules using a WASM runtime, and an API (i.e., WASI) for accessing OS resources of the underlying platform.
- Section 3 introduces the concept of data categorization and the use of various data protection techniques (e.g., data masking, redaction, etc.) to ensure the security of data in different domains or application scenarios using WASM modules, such as web traffic

data protection, API Security, microsegmentation, log traffic data protection, LLM traffic data protection, and integration with monitoring tools for the visualization of sensitive data flows.

- Section 4 presents a detailed security analysis of a WASM module by examining its development, deployment, and execution environment to ensure that the module satisfies the properties of a security kernel and can provide the necessary security assurance.
- Section 5 provides a summary of the topics covered in this document and discusses how WASM module functionality must continuously evolve to provide the security assurance needed to protect against data breaches and exfiltration in the context of increasingly sophisticated attacks on data.

2. Web Assembly Background

WebAssembly modules are deployed to protect data on microservices-based architectures in which the entire application (also called a cloud-native application because of its ubiquitous deployment in cloud and hybrid environments) consists of several distributed, loosely coupled, and independently scalable components called microservices. All services for this class of application (e.g., networking, security policies enforcement, state monitoring, configuration of runtime parameters) are provided by a centralized application-independent service infrastructure called the service mesh. This service mesh consists of a data plane that is primarily made up of proxies that house the various service modules. Using the family of APIs provided by the proxies, relevant service modules (e.g., network path determination) are implemented using the management/control plane of the service mesh. The WebAssembly is one such service module ecosystem implemented in the data plane proxies of a service mesh.

2.1. Origin

WASM modules originated in browser environments and were designed to run in memory-safe sandboxes, making them more secure than running client-side JavaScript. The execution model for running WebAssembly code in browsers is given in Appendix A. In addition to security, WASM modules have the following advantages [1]:

- **Performance:** Due to its low-level binary format targeted for modern processors, WASM modules provide near-native performance. Hence, it is considered the “fourth language” for the web alongside HTML, CSS, and JavaScript and is designed to enable high-performance applications in browsers.
- **Broad support:** It has broad accessibility and is supported in popular browsers, such as Chrome, Firefox, Edge, and Safari.

2.2. Progression Into Server-Side Environments

WASM modules progressed from browser to server environments when Mozilla introduced an open-source project called the WebAssembly System Interface (WASI) that provided a framework for WebAssembly apps to access operating system resources [4]. This allowed for content delivery networks (CDNs) to use WebAssembly to deploy customers’ apps without giving them access to the underlying CDN infrastructure.

2.2.1. Development and Deployment Process

The emergence of WASM compilers for several languages enabled developers to use their preferred languages to create server-side applications. Additionally, server-side WASM code could run inside containers as well as VMs. It is a potential candidate for SaaS-based offerings, just like VMs and containers. Its portability allows applications to run anywhere, making it an attractive option for various use cases.

The steps involved in developing a WASM module and running it using WASM runtime are [6]:

- **Source code writing:** Programs are written in languages (e.g., C++, C#, Rust, etc.) that have target WASM compilers available.
- **Parsing:** The code is parsed into an abstract syntax tree (AST) structure.
- **Compiling:** The code in AST structure is then compiled into a WASM module using AOT or JIT. The WASM module is generated in a binary format that can be executed by WASM runtime.
- **WASM runtime loading:** The WASM runtime loads the WASM module (with file name extension. wasm). If JIT is used, the compilation takes place after loading into WASM runtime.
- **Preparation for execution (i.e., instantiation):** The WASM runtime creates an executable instance from the WASM module by allocating memory, importing functions and objects, and establishing the execution environment for the module.
- **Code optimization:** During execution of the byte code, profiling is employed to identify frequently executed code, and a progressive optimization/re-optimization process takes place to gradually enhance performance until the code runs efficiently.

Figure 1 shows the ability to develop programs in different languages, convert them into WASM code, and run them under different processor architectures [4]. The execution model for WASM modules in the server environment and their comparison with the container execution model are described in Appendix B.

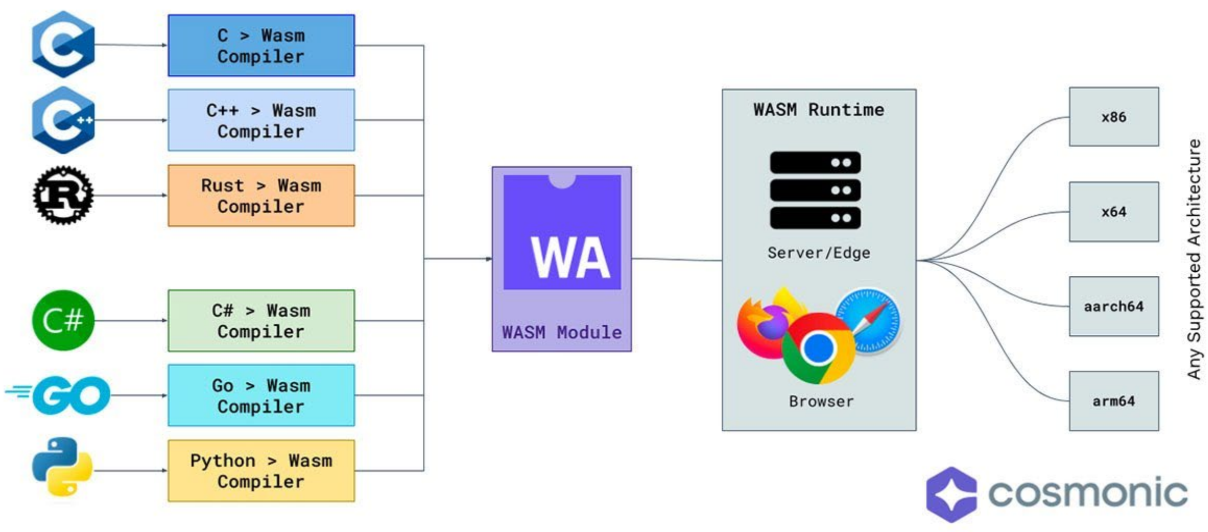

Fig. 1. Generating WASM modules and their execution [4]

2.3. Proxies as WASM Platforms

Proxies are increasingly being used as platforms for executing WASM modules. In cloud-native and microservices-based applications, proxies mediate inter-service communication. Open-source projects in proxies, such as Envoy, have extended their filter chain to allow for calling and executing WASM modules. These WASM modules can enforce policy-based authorizations or implement network resiliency measures, providing essential security controls for such applications. Additionally, the capabilities of these modules can be leveraged for data protection purposes.

The advantages of network based WASM modules include:

1. **Extensibility:** Proxies like Envoy can be extended with WASM modules, allowing developers to introduce custom logic and functionality without modifying the proxy's core codebase. This extensibility allows for the seamless integration of new features and capabilities.
2. **Security and isolation:** WASM modules run in a sandboxed environment, providing isolation from the host system and other modules. This isolation enhances security by preventing unauthorized access to system resources and mitigating the impact of potential vulnerabilities.
3. **Portability:** WebAssembly's portability ensures that WASM modules can run consistently across different proxy implementations and platforms, promoting a write-once, run-anywhere approach.
4. **Performance:** WASM modules can potentially offer better performance compared to the traditional scripting languages used for proxy extensions since they can be compiled to efficient machine code.
5. **Policy enforcement and network resiliency:** By executing WASM modules in proxies, organizations can enforce policies, implement authorization controls, and introduce network resiliency measures at the proxy level, ensuring consistent and centralized enforcement across distributed applications.
6. **Data protection:** WASM modules in proxies can be used to implement data filtering, transformation, or encryption mechanisms and ensure sensitive data protection as it flows through the proxy.
7. **Ecosystem and community:** The growing WebAssembly ecosystem and community provide libraries, tools, and resources that foster collaboration and accelerate the development of proxy extensions and data protection solutions.

As WASM continues to mature, its role in proxies will expand, enabling proxies to act as robust platforms for security and application logic execution. This evolution is particularly pertinent to data protection, which stands as a central theme of contemporary application development.

2.4. Proxy-WASM

Envoy Proxy, an open-source edge and service proxy, plays a pivotal role in managing the flow of traffic between microservices in many service mesh deployments. The collection of extensible APIs that it provides for various services is designated as xDS. An extension API that leverages the extensibility of these basic, foundational APIs of Envoy Proxy is the WebAssembly for Proxies (Proxy-WASM) runtime.

Proxy-WASM extends the adaptability of Envoy Proxy by enabling the deployment of WebAssembly modules within the proxy server. This integration allows for the execution of custom code directly within the proxy, providing a platform-independent and secure environment. The modularity of WebAssembly makes it an ideal choice for extending the functionalities of Envoy Proxy without the need for recompilation or significant changes to the existing infrastructure.

The architecture of Proxy-WASM within Envoy Proxy allows for the seamless integration and execution of custom logic at various stages of the request-response cycle. For example, a WASM module can intercept requests, inspect payload data, apply data categorizations, and redact data before proceeding. This level of granular control enhances the security posture of microservices architectures while maintaining performance and scalability.

Proxy-WASM can be leveraged to implement robust security measures for microservices communication. WASM modules can perform tasks, such as data categorization and mitigation directly within the proxy.

2.4.1. Role of WASM in Different Service Mesh Architectures

Service mesh architectures have traditionally utilized sidecar proxies, which are implemented as containers and deployed alongside each service within a Kubernetes pod. These sidecar proxies manage both inbound and outbound traffic for their respective services, creating an ideal WASM-based insertion point for in-transit categorization.

Additionally, newer architectural patterns (e.g., proxy implementation/deployment models) recognize that sidecar proxies are excessive because many services do not have L7-level services. The ambient waypoint proxy pattern seeks to simplify the sidecar model by centralizing and simplifying traffic management and policy enforcement. In this pattern, waypoint proxies are deployed at the node level, which provides application services either per namespace or per service account. They manage all ingress and egress traffic for the services within their designated scope. In both proxy deployment models, the WebAssembly VM intercepts and analyzes traffic in the exact same way, providing a transparent deployment for WASM-based data categorization policies and modules.

Outside of traditional Envoy-based service mesh proxies, there are several runtime environments where WASM modules can be deployed to classify sensitive data in transit. Many API gateways now support WASM along with commercial content delivery network (CDN) platforms, such as Fastly’s WASM Compute Platform and Cloudflare’s WASM Workers.

2.5. WASI-HTTP

With its application binary interface (ABI), Proxy-WASM facilitates communication between WebAssembly modules and host environments, specifically proxies. It has a mature specification adopted by various proxy servers and traces its origins to efforts within the Envoy project to extend the capabilities of proxy servers using WebAssembly. Proxy-WASM ensures that extensions written for one proxy can be reused in others, promoting a write-once, run-anywhere approach. Proxy-WASM's ABI and event-driven streaming APIs have been incorporated into several production-level proxies, demonstrating the project's practical application and influence.

In contrast, WASI-HTTP — a WASM-based API — has evolved through iterations to define interfaces for handling HTTP requests and responses directly within WASM modules. It aims to provide a minimal and streamlined execution environment for WebAssembly-based HTTP proxies and is designed to seamlessly integrate with existing web infrastructure, such as service workers and reverse proxies, without requiring a complex runtime system. WASI-HTTP is already in production in some environments and supports scalable and dynamic WASM instance creation in response to web traffic, laying the groundwork for future innovations like linking HTTP intermediaries through the component model.

Both WASI-HTTP and Proxy-WASM are shaping the landscape of WebAssembly in networked and distributed systems. While WASI-HTTP is allowing for simplified HTTP communication within WebAssembly applications, Proxy-WASM exemplifies the successful implementation of a standardized interface across multiple proxy implementations. Their collaborative development highlights a symbiotic relationship, with WASI-HTTP potentially leveraging Proxy-WASM's ABI to further enhance the capabilities and reach of WebAssembly in networking scenarios.

2.6. eBPF

Using WASM to parse human-readable text in Layers 4–7 offers several advantages over technologies like eBPF, particularly regarding handling complex application-layer data, such as HTTP. While eBPF is powerful for data capture and manipulation directly within the kernel, its use for parsing detailed HTTP traffic can be complex and potentially excessive for some applications. This complexity stems from the need to handle the intricacies of HTTP within the kernel — a task that can restrict performance and introduce security concerns if not managed correctly. Additionally, eBPF imposes numerous restrictions and requires extra effort for data processing and general-purpose computation.

WASM provides a secure, sandboxed environment that is suitable for efficiently executing code across multiple platforms and parsing application-layer protocols. WASM can be used in user spaces and server environments, allow easier integration with existing parsing libraries and tools, reduce complexity, and potentially enhance the reliability of parsing operations. Its portability and ability to embed in various runtime environments make it a practical choice for network traffic analysis tasks, including those involving protocols that handle human-readable text.

3. Data Protection in Transit

One of the first and most fundamental tasks in data protection is classifying data to identify the need for further operations (e.g., sanitization, filtering, etc.).

3.1. Data Categorization Techniques

Data in transit can vary wildly between structured and unstructured formats. For real-time categorization and protection, care must be taken to formulate the right approach. The performance of each categorization event is critical to ensuring that minimal latency is added as the process takes place. By executing WASM modules in proxies, organizations can implement data categorization and filtering mechanisms at the proxy level. This approach allows for the identification and protection of sensitive data as it flows between services.

Regex and ML models can be used within these WASM modules to detect patterns and classify data in real time, enabling the implementation of appropriate data protection measures, such as redaction, encryption, or access control policies. Regex matching can identify complex patterns for nuanced categorization schemes, and ML tools can detect patterns that signify categorization attributes. This latter process involves classifying a set of example data and training one or more models to analyze and classify future data. Though it is potentially the most effective automatic categorization method, it requires significant setup and management. The training data sets must be comprehensive to provide ample information for accurate categorization detection.

Unlike other data categorization techniques that operate on data at rest, in-transit categorization provides the added dimension of time as traffic is analyzed. When combining data categorization with the time it was accessed or sent, data flows can be visualized and understood. Once models have been trained on normal data flow patterns, it becomes clear when a violation in data access has occurred or when an unpermitted data flow has been established. By leveraging the capabilities of WASM modules in proxies, organizations can gain visibility into data flows, detect anomalies, and take proactive measures to protect sensitive data as it moves between services in cloud-native and microservices-based applications.

3.2. Techniques for Data Protection

This section describes the practical uses of the data protection techniques dynamic data masking (DDM), user and entity behavior analytics (UEBA), and data loss prevention (DLP) within WASM modules in various application scenarios with a focus on the domain data that pertains to each application scenario.

3.2.1. Web Traffic Data Protection

In-transit data categorization across web protocols like HTTP/2 and gRPC enable the observability of data flows between services and clients. By classifying data in motion, organizations can monitor how sensitive information is accessed by both unauthenticated and authenticated identities. WASM modules can use regex and ML models to identify sensitive data patterns in HTTP payloads and redact, mask, or block classified data transmissions based on configured policies. Example applications include:

- **E-commerce websites:** Monitoring credit card details and personal information during transactions to ensure that they are properly encrypted and masked, preventing unauthorized access.
- **Healthcare applications:** Protecting patient data by detecting and encrypting sensitive information, such as medical records and personal identifiers before they are transmitted between systems.
- **Corporate communications:** Scanning and securing internal emails and messages to prevent data breaches and ensure compliance with internal data protection policies.

3.2.2. API Security

APIs are critical conduits for sensitive data and are often targeted for attacks. Monitoring data transmitted to and from APIs is essential for detecting vulnerabilities, such as application-level DDoS attacks, SQL injection, or data exfiltration. Many API gateways and service meshes support running WASM modules for enhanced security. These modules can implement authentication, rate limiting, and payload inspection for API traffic. Example applications include:

- **Financial services:** Protecting API endpoints that handle financial transactions by detecting and blocking SQL injection attempts and unauthorized access attempts.
- **Social media platforms:** Monitoring data flow through APIs to prevent the exfiltration of user data and ensure that sensitive information, such as login credentials and personal messages, are protected.
- **IoT devices:** Securing data transmitted from IoT devices to backend systems and detecting anomalies in data patterns that might indicate a security breach.

3.2.3. Microsegmentation

In microsegmentation, in-transit data categorization enhances asset inventory reporting. This advanced categorization enables organizations to identify and track critical assets and their data flows to ensure alignment with data protection policies. This granular insight is especially valuable for assets that handle PII or financial data, bolstering data governance and compliance efforts.

While Kubernetes (K8s) networking policies offer segmentation, managing and testing these policies can be resource intensive. Traditional network policies rely on static rule sets that require meticulous configuration and maintenance. Comprehensive testing across dynamic environments poses operational challenges, and these policies lack granular visibility into data content, making it difficult to accurately differentiate between legitimate and malicious traffic.

In contrast, in-transit data categorization offers a dynamic and granular approach. By analyzing data flows in real time, organizations gain actionable insights into the content and context of network traffic. This enables the precise enforcement of security controls based on data attributes, such as sensitivity levels or compliance requirements. Example applications include:

- **Financial institutions:** Implementing microsegmentation to protect critical systems that handle transaction processing to ensure that only authorized services can access sensitive financial data.
- **Healthcare providers:** Segregating networks within a hospital to ensure that medical devices and patient data systems are isolated from less secure administrative networks.
- **Retail chains:** Using real-time data categorization to manage data flows between point-of-sale systems and backend inventory systems to prevent unauthorized access to sales data and customer information.

3.2.4. Log Traffic Data Protection

Regulated organizations often face the challenge of sensitive data leaking into log streams. Since all log protocols operate at Layer 4 and traverse service proxies within a service mesh, addressing potential leaks at their source allows organizations to secure data before it disperses into various storage systems, effectively mitigating the risk of exposure.

Example applications include:

- **Financial services:** Ensuring that transaction logs do not contain unmasked credit card numbers or personal identification information to prevent accidental leaks.
- **Healthcare providers:** Protecting patient data in system logs by redacting sensitive information before it is stored or transmitted to logging systems.
- **E-commerce platforms:** Monitoring and sanitizing log data to prevent the exposure of customer order details and personal information.

3.2.5. LLM Traffic Data Protection

Due to their scalability needs, large language models (LLMs) typically operate within service mesh architectures. Classifying both prompt and response data in transit is crucial for governance. This enables organizations to maintain visibility over the data flows of deployed LLMs and ensure compliance with regulatory standards and organizational policies for data protection.

Example applications include:

- **Customer support systems:** Monitoring interactions between customers and automated support bots to ensure that sensitive customer data is not inadvertently exposed or logged.

- **Content Moderation:** Ensuring that data processed by LLMs for content moderation is handled in compliance with privacy regulations to protect user information.
- **Data Analysis Services:** Classifying and securing data used by LLMs in analytics platforms to prevent unauthorized access to sensitive business insights and customer data.

3.2.6. Credit Card-Related Data Protection

WASM modules are also used to protect data related to credit card transactions, as laid out in PCI DSS 4.0 specifications. This is achieved by incorporating the following functions into WASM modules:

- Clearly identify and document all areas in which sensitive data (e.g., cardholder data, authentication values, encryption keys, etc.) is stored, processed, or transmitted. This includes databases, servers, applications, and network segments that handle card holder data.
- Generate data-flow diagrams or other technical or topological solutions that identify flows of account data across systems and networks.
- Identify all data flows for the various stages of payment transactions (e.g., authorization, capture settlement, chargebacks, and refunds) and acceptance channels (e.g., card present, card not present, and e-commerce).

3.2.7. Monitoring Tools to Visualize Sensitive Data Flows

WASM modules can also be programmed to collect and emit metrics and telemetry data in various formats to monitoring tools that are used to visualize the flow of sensitive data (e.g., Prometheus, Grafana etc.). By examining the normal rate of sensitive data flow over time, visual indicators, such as spikes, can be used to identify data leakage incidents and unauthorized data exposures. Subsequent investigations can then ensure compliance with data protection regulations and reduce the risk of continued data breaches.

4. Security Analysis of WASM Modules

To realize the security goals for which the WASM modules are deployed, the whole ecosystem under which these modules execute must obey the properties of a security kernel:

1. It is always invoked (i.e., non-bypassable).
2. It is small and verifiable.

Consider the satisfaction of the first property in the context of two proxy implementation models in a service mesh. In the sidecar proxy model, a proxy is implemented as a container that coexists with each microservice in the same pod and runs in the same network space as the service. All traffic coming into and emanating from the microservice must pass through the proxy and the applications running inside of the proxy. Hence, the WASM module that provides the data protection function deployed inside the proxy will always be invoked.

In the ambient proxy implementation model, the network link to a service or group of services associated with a namespace has to pass through the node hosting the waypoint proxy serving that service or group of services for a designated namespace. No direct network paths to the service or group of services exists. Again, the WASM module provides data protection for services under the scope of the proxy has to be invoked.

To meet the second property of the security kernel (i.e., the security is verifiable), a security analysis of the entire execution environment for the WASM modules must be performed. The life cycle of a WASM module begins with a source code in some supported language (e.g., C, C++, or Rust) that is then compiled using a target compiler (e.g., using LLVM) into a binary byte code that is run by a runtime module (i.e., WASM runtime). Access to the operating system or host resources is enabled by calling a module that implements an API called WASM System Interface (WASI).

The security analysis of the WebAssembly ecosystem can be considered in terms of the following topics:

1. WASM security goals and security feature sets
2. Memory model and memory safety
3. Execution model and control flow integrity
4. Security of API access to OS/host resources
5. Protection against side-channel attacks
6. Protection against injection attacks
7. Deployment and operating safety

4.1. WASM Security Goals and Security Feature Sets

The WASM security model has two important goals: (1) protect *users* from buggy or malicious modules, and (2) provide *developers* with useful primitives and mitigations for developing safe applications within the constraints of (1)[8].

4.1.1. User-Level Security Features

Each WASM module executes within a sandboxed environment that is separated from the host runtime using fault isolation techniques. This implies that:

- Applications execute independently and cannot escape the sandbox without going through appropriate APIs.
- Applications generally execute deterministically with limited exceptions.

Additionally, each module is subject to the security policies of its embedding. Within a web browser, this includes restrictions on information flow through same-origin policy. On a non-web platform, this could include the POSIX security model.

4.1.2. Security Primitives for Developers

The design of WebAssembly promotes safe programs by eliminating dangerous features from its execution semantics while maintaining compatibility with programs written for C/C++. Modules must declare all accessible functions and their associated types at load time, even when dynamic linking is used. This allows for the implicit enforcement of control-flow integrity (CFI) through structured control flow. Since compiled code is immutable and not observable at runtime, WebAssembly programs are protected from control flow hijacking attacks.

- Function calls must specify the index of a target that corresponds to a valid entry in the function index space or table index space.
- Indirect function calls are subject to a type of signature check at runtime, and the type signature of the selected indirect function must match the type signature specified at the call site.
- A protected call stack that is invulnerable to buffer overflows in the module heap ensures safe function returns.
- Branches must point to valid destinations within the enclosing function.

4.2. Memory Model and Memory Safety

As there are only four primary data types defined by WASM, compilers targeting WASM implement their own stack in an area called linear memory, which becomes the main memory of a WASM program. A linear memory is a contiguous, byte-addressable range of memory that can be considered as an untyped array of bytes. This enables the program to store non-scalar data and any variable whose address needs to be taken by the module [10]. In addition to linear memory, there is the code space, execution stack, and runtime data structure [11]. The execution stack mainly stores local variables, global variables, and return addresses.

Compilers that target WASM also create an area for the heap in the linear memory. This area is reserved at the end of the linear memory so that it can dynamically grow when additional space is allocated for the linear memory. This linear memory is sandboxed — disjointed from code space, execution stack, and runtime data structure [11] — and prevents WASM modules from accessing other memory areas. These other memory regions are isolated from the internal memory of the runtime and are set to zero by default unless otherwise initialized. However, modules can access the data stored on the execution stack via dedicated instructions. The actual data address on the execution stack is never shown to the module. A compliant runtime ensures that the module does not break WASM's memory model [12]. This is done by bounds-checking access to the linear memory at the region level. If the module accesses the memory outside of the linear memory, the program traps and prevents modules from accessing data outside of their allocated memory [11].

Another common class of memory safety error involves unsafe pointer usage and undefined behavior. This includes dereferencing pointers to unallocated memory (e.g., NULL) or freed memory allocations. In WebAssembly, the semantics of pointers have been eliminated for function calls and variables with a fixed static scope, allowing references to invalid indexes in any index space to trigger a validation error at load time or — at worst — a trap at runtime.

However, the bounds-checking process is performed at the level of the linear memory, and modules can access the entire linear memory without restriction. Linear memory is not protected by standard techniques like stack canaries or guard pages. Therefore, buffer overflows — which occur when data exceeds the boundaries of an object and accesses adjacent memory regions — cannot affect local or global variables stored in index space. Data stored in linear memory can also overwrite adjacent objects since bounds-checking is performed at linear memory region granularity and is not context-sensitive.

4.3. Execution Model and Control Flow Integrity

WASM code is executed when instantiating a module or when an exported function is invoked on a given instance [12]. The execution behavior of a WASM module is defined in terms of an abstract machine that models the program state. This abstract machine includes a stack that records the operand values and control constructs as well as an abstract store that contains the global state.

WASM primarily achieves control flow integrity through the execution semantics of the language itself. The definition of the WASM bytecode [12] limits the constructs that are possible to express. It defines valid code constructs and how control flow may only jump to the beginning of a valid construct. Arbitrary jumps (e.g., goto statements) are not allowed; only structured control flow is provided. Consequently, a grammatically valid WASM module can only jump to the beginning of valid constructs (e.g., conditional constructs or functions) [11].

An additional factor contributing to the control flow integrity is the prevention of call redirection through restrictions on indirect function calls. Restrictions are applied regarding functions that the module can indirectly call. To indirectly call a function, the module provides a runtime index to a table. This table holds the signatures of the functions that the module defines or imports and that can be indirectly called. When an indirect call is made, the runtime checks that the calling signature and the signature of the called function match. If there is a type mismatch or an out-of-bounds table access, a trap occurs [11] .

4.4. Security of API Access to OS and Host Resources

By default, WASM does not have access to the resources of the host (e.g., file system, network, system calls). Modules can import externally defined functions provided by the host or other modules. APIs common to many use cases are currently being standardized in the WASI [13]. The capability-based security model of WASI enables the introduction of a verified secure runtime system, as shown in [14].

4.5. Protection From Side-Channel Attacks

The WASM language specification [12] clearly states that side-channel attacks are to be addressed by the runtime. Currently, Wasmtime implements a few forms of Spectre mitigations. Bounds checks for the runtime index used in indirect calls and some other instructions are mitigated to ensure that speculation goes to a deterministic place [15]. However, some side-channel attacks can occur, such as timing attacks against modules.

In the future, additional protections may be provided by runtimes or the toolchain, such as code diversification or memory randomization like addressing space layout randomization (ASLR) or bounded pointers (i.e., “fat” pointers).

4.6. Protection Against Code Injection and Other Attacks

Control-flow integrity and protected call stacks prevent direct code injection attacks. Thus, common mitigations, such as data execution prevention (DEP) and stack smashing protection (SSP), are not needed by WASM programs. Nevertheless, other classes of bugs are not obviated by the semantics of WebAssembly. Although attackers cannot perform direct code injection attacks, it is possible to hijack the control flow of a module using code reuse attacks against indirect calls. However, conventional return-oriented programming (ROP) attacks using short sequences of instructions (i.e., “gadgets”) are not possible in WebAssembly because control-flow integrity ensures that call targets are valid functions declared at load time. Likewise, race conditions, such as time-of-check to time-of-use (TOCTOU) vulnerabilities, are possible in WebAssembly since no execution or scheduling guarantees are provided beyond in-order execution. Yet another security limitation is that there are no audit tools to track the changes made by WASM modules.

4.7. Deployment and Operating Security

The security features described so far pertaining to run time security. The following capabilities relate to the controls that are present for deployment and integrity of operations.

- The ability to create the WASM filter in the proxy can be controlled through the native access mechanism in the service mesh (e.g., RBAC).
- Only calls using HTTP and gRPC protocols are allowed.

- Even for making those calls, only clusters known to the proxy can be used. Similarly, responses coming from clusters already known to the proxy are examined.

5. Summary and Conclusions

This document describes how WASM modules can be developed and deployed in service mesh proxies for the real-time protection of data in transit in cloud-native application architectures. Various data protection techniques can also be used to protect data in different domains of various application scenarios. WASM modules can provide telemetry data for monitoring tools that provide visual images of sensitive data flows. A detailed security analysis of the WASM module development, deployment, and execution environment can ensure that necessary security assurances are obtained by running the modules as part of the application infrastructure environment (e.g., in service mesh proxies).

The data categorization and protection techniques built into WASM modules must continuously evolve to keep pace with increasingly sophisticated attacks on data that result in new forms of data breaches, data leakages, and other forms of data exfiltration.

References

[1] Doerrfeld B (2023) Wasm: The Next Generation Beyond Kubernetes? Available at https://cloudnativenow.com/features/wasm-the-next-generation-beyond-kubernetes/?utm_medium=email&_hsmi=293085224&_hsenc=p2ANqtz-9qlZw1MWD8-AHNH54OoPyWB7vOLe0uG0KZSIe2uH1sbXAD_rmmhyXHMThd0GMdMLUb-w8axL_Gv1L2RM9Nq55L2eCysg&utm_content=293086040&utm_source=hs_email

[2] Krasnov M (2020) Web Assembly is the End of Internet as we know it. Available at https://betterprogramming.pub/webassembly-is-the-end-of-the-internet-as-we-know-it-9085a49cbc7b

[3] WebAssembly (2024) WebAssembly Concepts. Available at https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts#see_also

[4] TechTarget (2022) Server-side WebAssembly prepares for takeoff in 2023. Available at https://www.techtarget.com/searchitoperations/news/252527414/Server-side-WebAssembly-prepares-for-takeoff-in-2023

[5] Medium (2023) WASM and Kubernetes – A new era of application development. Available at https://medium.com/@seifeddinerajhi/wasm-and-kubernetes-a-new-era-of-cloud-native-application-deployment-b3c59b39f640

[6] Podobnik TJ (2023) WASM Runtimes Vs Containers: Cold Start Delays (Part 1). Available at https://levelup.gitconnected.com/wasm-runtimes-vs-containers-performance-evaluation-part-1-454cada7da0b

[7] ITPro (2024) WASM Today, AI Tomorrow: KubeCon Extends its Reach. Available at https://www.itprotoday.com/cloud-computing-and-edge-computing/wasm-today-ai-tomorrow-kubecon-expands-its-reach?_mc=NL_DR_EDT___20240401&cid=NL_DR_EDT___20240401&utm_rid=CPNET000059406774&utm_campaign=57716&utm_medium=email&elq2=a6cba5014e5b49bb9a1fe0c3e0351bd2&sp_eh=87aea8874bbd0a1985055c93c957744c11570c6718777eca378db1b4436de815

[8] Security.md (2018) WebAssembly Security. Available at https://github.com/WebAssembly/design/blob/main/Security.md

[9] Huang W, Paradies M (2021) An Evaluation of WebAssembly and eBPF as Offloading Mechanisms in the Context of Computational Storage. Available at https://marcusparadies.github.io/files/ebpf_vs_wasm_report.pdf

[10] Daniel Lehmann D, Kinder J, and Pradel M. (2020). Everything Old is New Again: Binary Security of WebAssembly. In USENIX Security

[11] Haas A., et all (2017). Bringing the web up to speed with WebAssembly. In PLDI.

[12] WebAssembly Community Group (2023). WebAssembly Specification. Draft Release 2.0 (Draft 2023-04-24). Available at https://webassembly.github.io/spec/

[13] WebAssembly Community Group (2023). WebAssembly System Interface. Available at https://github.com/WebAssembly/WASI

[14] Johnson E., et all (2023). WaVe: A verifiably secure WebAssembly sandboxing runtime. In Proceedings of IEEE Symposium on Security and Privacy (SP).

[15] Wasmtime (2023). Security - Wasmtime. Available at https://docs.wasmtime.dev/security.html

Appendix A. Execution Model for Web Assembly in Browsers

WASM runtime originated with browsers that enabled the running of native code (i.e., code written in low-level languages such as C, C++, Rust, etc.).

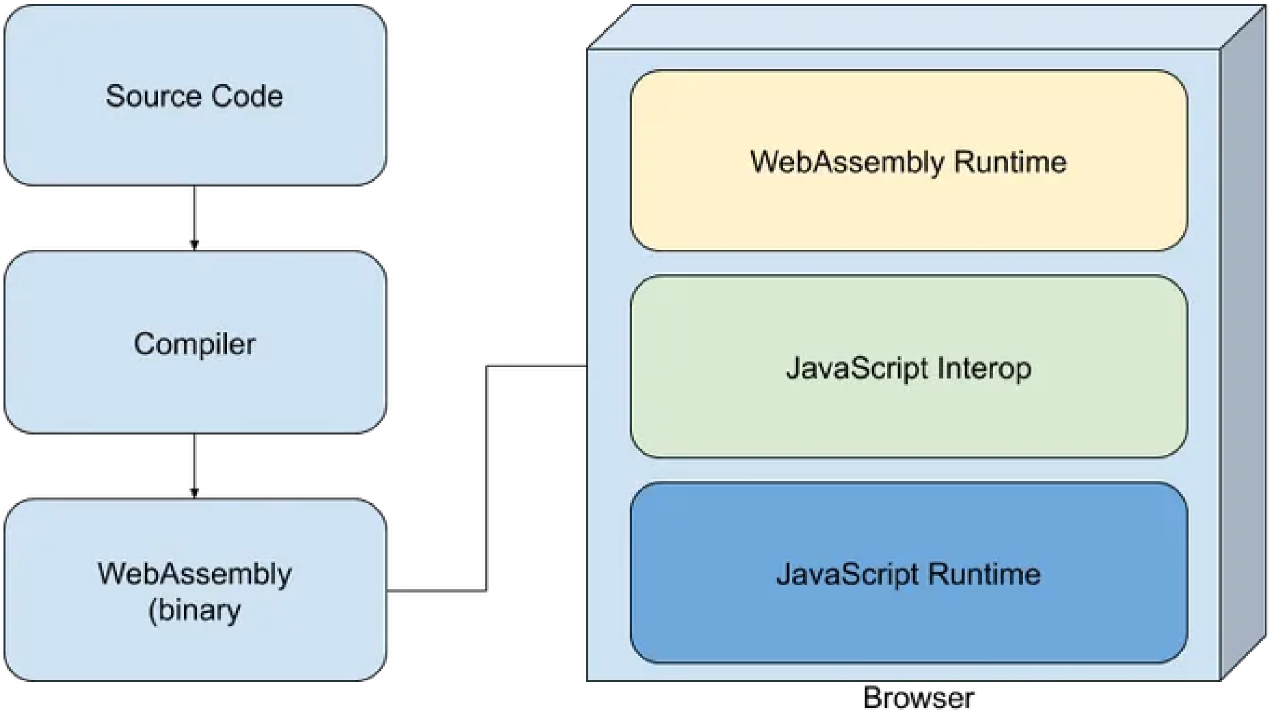

Fig. 2. WASM Module Development & Execution in Browsers

The WebAssembly program is run through a compiler (also called a WebAssembly target compiler) that inputs code into an LLVM-compliant language and produces a binary .wasm file. That file is loaded onto the existing JavaScript code by the JavaScript Interop layer and executed by the WebAssembly runtime [2]. The .wasm file is a low-level assembly language file in binary format.

The WASM compiler for C, C++, and Rust takes the source code written in those languages and compiles it into a WASM module. Then the necessary JavaScript "glue" code is generated for loading and running the module and an HTML document is used to display the results of the code. The details of this process are explained in [3].

Appendix B. Comparison of Execution Models for Containers and WASM Modules

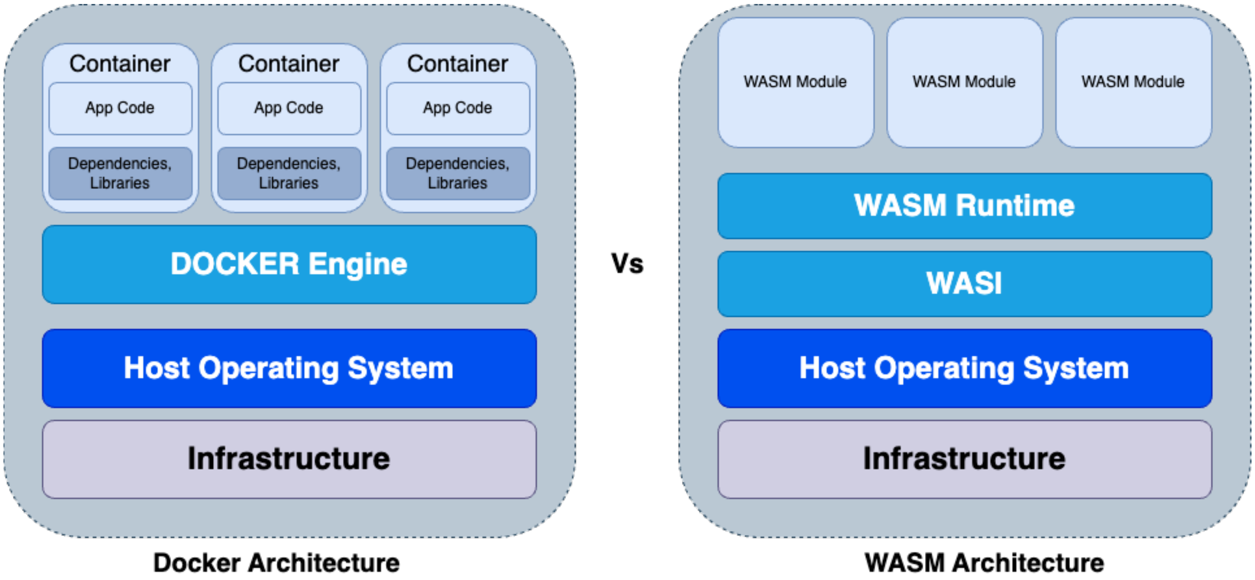

Fig. 3. Comparison of Execution Stack for Containers & WASM Modules

Container images are created by combining the program containing the application logic with its dependencies (e.g., runtime libraries) in a container runtime (e.g., docker). The container is a full file system (i.e., utilities, binary), and the generated image should be for a designated OS kernel and processor architecture (e.g., Intel, Arm, etc.). For example, if a Raspberry Pi OS is running a docker image, then an image for the C/C++ application based on a Linux image must be created and compiled for the ARM processor architecture. Otherwise, then container will not run as expected [5].

In contrast, WASM modules and binaries are precompiled C/C++ applications that do not rely on being coupled with a host OS or system architecture because they do not contain a precompiled file system or low-level OS primitives. Every directory and system resource is attached to a WASM module during runtime facilitated by WASI and then run using WASM runtime. In other words, WASI is used to access all resources under the control of the OS, essentially decoupling the code from its dependency on the platform architecture.