Beyond Container CVE Analysis: A GitOps-Based Attestation and Sandbox Framework for Container Supply Chains

Evangelos Syrmos*, Panagiotis Radoglou-Grammatikis*‖, Efklidis Katsaros*, Jyoti Sekhar Banerjee‡, Anastasia Kazakli*, Konstandinos Panitsidis§, Vasileios Vitsas¶ and Panagiotis Sarigiannidis‖

***Abstract*—As software supply chain accelerate through DevOps automation and continuous delivery, container images have become the primary vector for both application development and security compromise. While static vulnerability scanners can detect known CVEs, they are unable to uncover zero-day malware or runtime threats- particularly in container images sourced from public registries, which are maintained by individual contributors of varying intent and trustworthiness. In this paper, we introduce a GitOps-drive sandboxing framework for proactive and tamper-resistant container image attestation, addressing the urgent need for deeper analysis before deployment. Our approach combines static vulnerability detection with dynamic behavioral inspection using gVisor-based sandboxing. Through filesystem analysis, system call tracing, and network activity monitoring, the framework identifies malicious patterns and anomalies. Adopting this framework will empower developers and security teams to enforce stronger trust guarantees even in the absence of SBOMs or SLSA levels. This framework lays the foundations for a resilient, trustworthy software delivery at scale in compliance with NIST SP 800-218 and ISO/IEC 27001 standards.**

***Index Terms*—docker, oci, docker containers, devops, cybersecurity, malware detection, tracing, cloud native security**

I. Introduction

Containerization has become an integral part of modern Continuous Integration and Continuous Delivery/Deployment (CI/CD) pipelines, providing portability and consistency across heterogeneous environments [1]. Yet, reliance on public contributed container images introduces significant security risks: static vulnerability scanners can only identify known Common Vulnerabilities and Exposures (CVEs), leaving zero-day exploits and especially runtime-only malware undetected. Container registies such as Docker Hub, Github Container Registry, and Quay perform automated CVE scanning in the background, as a preventative measure. However, they ignore to capture the runtime behavior, where malicious artifacts can still traverse into production. To this end, hardend base images reduce the CVE surface by finetuning the dependency graphs of known container images. Altough this approach significantly reduces the CVE exposure, they are not able to guarantee runtime safety when adversaries embed payloads that trigger under specific conditions.

To address this gap, we introduce a GitOps-based sandboxing framework that combines industry-standard static CVE scanning with dynamic sandboxing inspection (i.e., filesystem access, syscall tracing, network monitoring). By aggregating CVE summaries, external STIX threat reports, and runtime logs into a unified attestation, our approach allows Software Engineers to validate container images even in cases where Software Bills of Materials (SBOMs) or Supply Chain Levels for Software Artifacts (SLSA) metadata are absent, while conforming to the NIST SP 800-218 and ISO/IEC 27001 standards [2], [3].

The contribution of this paper is summarised as follows:

1) A GitOps-compatible attestation framework is presented that combines static and runtime container analysis.
2) A two-layer sandboxing architecture is introduced to securely execute and monitor untrusted container images.
3) We demonstrate how runtime monitoring of system calls and network activity can enhance the detection of previously unknown threats in the software supply chain.

The remainder of this paper is organized as follows. Section II performs a background review, while Section III dives into the related work on container security and sandboxing methods. Section IV details the architecture and components of the proposed framework using the C4 Model abstraction, while Section V concludes the paper with directions and future work.

*This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101093069 (P2CODE). Disclaimer: Funded by the European Union. Views and opinions expressed are, however, those of the authors only and do not necessarily reflect those of the European Union or European Commission. Neither the European Union nor the European Commission can be held responsible for them.

* E. Syrmos, P. Radoglou-Grammatikis, E. Katsaros and A. Kazakli are with K3Y, Studentski District, Vitosha Quarter, Bl. 9, 1700 Sofia, Bulgaria – email: {esyrmos, pradoglou, ekatsaros, nkazakli}@k3y.bg

‡ J. S. BanerjeeV is with the Department of Computer Science and Engineering (AI & ML), Bengal Institute of Technology, Kolkata, India and Remote Researcher, Internet of Things & Applications Lab (ITHACA), University of Western Macedonia, Greece – email: tojyoti2001@yahoo.co.i

§ K. Panytsidis is with the Department of Management Science & Technology, University of Western Macedonia, 50100, Kozani, Greece – email: kpanytsidis@uowm.gr

¶ V. Vitsas is with the Department of Information and Electronic Systems Engineering, International Hellenic University, 57400 Thessaloniki, Greece – email: vitsas@it.teithe.gr

‖ P. Radoglou-Grammatikis and P. Sarigiannidis are with the Department of Electrical and Computer Engineering, University of Western Macedonia, Active Urban Planning Zone, Kozani, 50100, Kozani, Greece – email: {pradoglou, psarigiannidis}@uowm.gr

II. Background

A. Container Security Challenges

The adoption of containerization technologies like Docker has transformed software development, enabling continuous integration and continuous delivery (CI/CD) pipelines. Regardless of their advantages, containers introduce critical security challenges. Containers occasionally interact with the host operating system kernel, increasing their attack surfaces. In detail, common threats include base-image vulnerabilities, misconfigurations, and software supply chain compromises [4], [5]. Public container registries such as Docker Hub, GitHub Container Registry, and Quay.io are particularly vulnerable due to varying degrees of trustworthiness of contributors [6]. Attackers often exploit this level of trust by disguising malicious images as legitimate, introducing backdoors or injecting malware during the image build process or via third-party dependencies [7]. Misconfigured containers running with excessive privileges can further introduce lateral movement or host compromise [8].

B. Limitations of Static Analysis

Static vulnerability scanning tools such as Trivy, Clair, and Grype are commonly used to identify known Common Vulnerabilities and Exposures (CVEs) by matching container software components against public vulnerability databases [9]–[11]. Although these tools are effective for detecting documented vulnerabilities, they fail to detect zero-day threats, runtime-triggered exploits, or sophisticated malware embedded within container filesystems. Static analysis alone thus provides a false sense of security, as it lacks the capability to detect threats that activate under specific runtime conditions [12], [13]. Therefore, static analysis should be viewed as foundational yet insufficient on its own, necessitating complementary security methods to address advances and hidden threats.

C. Dynamic Analysis and Sandboxing Fundamentals

Dynamic analysis complements static approaches by observing container behaviour during runtime to detect anomalous activities. Sandboxing, a form of dynamic analysis, provides secure environments where untrusted container images can be executed safely. Such sandboxing technologies include Google's gVisor, which intercepts system calls at the user-space kernel, isolating containers from the host kernel, thereby significantly reducing the risk of container escapes [14], [15]. On the other hand, Kata containers use micro-virtualization for hardware-based isolation. These dynamic techniques can capture runtime anomalies, such as suspicious system calls or unauthorised network activity, which are typically indicative of embedded malware or unknown vulnerabilities [16].

III. Related Work

A. Existing Static Vulnerability Scanners

A considerable work has been focused on the static analysis of container images to detect unknown vulnerabilities. Tools such as Trive, Clair and Grype implement signature-based scanning techniques that match image contents against public vulnerability databases, including the National Vulnerability Database (NVD). These tools are widely adopted within CI/CD pipelines due to the simplicity, scalability, and effectiveness in identifying documented CVEs.

Nevertheless, static analysis exhibits fundamental limitations when considering the sophistication of contemporary attackers in coordinated large-scale attacks. Specifically, they are incapable of identifying obfuscated threats, polymorphic malware, and zero-day exploits that are not yet included in CVE databases. Furthermore, static scanners are not designed to capture contextual behaviours that manifest only during runtime, such as conditional execution of dormant payloads or lateral network probing. As a result, reliance on static scanners alone may lead to a false sense of security, particularly when container images originate from unverified sources or public registries maintained by contributors of unknown trustworthiness.

B. Runtime Analysis and Sandboxing

To tackle the limitations of static scanners, studies have proposed dynamic and behaviour-based analysis approaches that require precise scaffolding of infrastructure to capture it. gVisor, developed by Google, introduces a user-space kernel that intercepts and emulates system calls, thereby providing a strong isolation between running container workloads and the host system. Similarly, Kata Containers leverage micro-virtualisation to run each container in a lightweight virtual machine, offering hardware-enforced security boundaries.

In addition to isolation, system call tracking, file system interaction monitoring, and network traffic analysis have been explored as detection mechanisms. Specifically, the COSOCO malware detector, which employs a novel representation of container images as RGB-encoded tarball visualisations, enabling the classification through convolutional neural networks [17]. Although such techniques demonstrate promise in identifying unknown or obfuscated threats, they are frequently deployed in an ad hoc manner, lack interoperability with standard reporting formats and are detached from DevOps-native workflows.

Concerning is the fact that current sandboxing approaches do not consistently integrate with automated software delivery practices. This absence of formalised attestation artifacts hinders their application in high-assurance environments that require traceability and auditability of runtime behaviours.

Specifically, multiple classes of runtime exploits continue to challenge containerised environments. These include container escape vulnerabilities, cross-container attacks, and fileless malware injections. Notably, the RunC runtime vulnerability (CVE-2019-5736) enabled attackers to gain host-level access by exploiting weaknesses in how system calls were handled, allowing a compromised container to overwrite the host binary [18]. Translating this into orchestrated environments such as Kubernetes, makes this CVE more critical. Since breaches in

one container can result in lateral movement across pods via shared resources.

C. Secure Build Pipelines and Industry Standards

Recent industry efforts have sought to improve software supply chaing integrity by promoting verifiable build processes and artifact provenance. For instance, Docker and Chainguard offer minimal, hardened base images that are engineered to reduce the CVE surface area. Simultaneously, cryptographic signing and verification of container images are being realised by the Sigstore project, thereby ensuring immutability and origin of authenticity.

The SLSA framework further defines the best practices for software build and distribution pipelines. For instance, SLSA Level 3, the systems are expected to enforce strong guarantees on build provenance through authenticated CI/CD workflows, isolated builders, and tamper-evident logs. Although this level significantly increased the artifact trustworthiness barrier, SLSA L3 remains focused on the integrity of the build process rather than the behaviour of the artifact post-deployment.

In addition, the cryptographically verified artifacts do not ensure the possibility of embedded malicious behaviour that activates only under specific runtime conditions. An underestimated portion of publicly available container images that lack SBOMs remain unclassified under any formal trust model [19].

Consequently, there is an urgent need for complementary mechanisms that need to be adopted to provide behavioural attestation of container images as sophisticated adversaries find new ways to inflict catastrophic downtimes. The proposed framework addresses this gap with a retroactive attestation approach, which enhances the security posture even when SBOMs or SLSA levels are absent. Ultimately, this design is intended to support compliance security standards such as NIST SP 800-218 and ISO/IEC 27001, which emphasize software supply chain integrity and verifiable artifact provenance.

IV. Architecture & Specifications

To realize a proactive malware detection in containerized CI/CD pipelines, the proposed framework adopts a layered architecture modeled using the C4 Model, which captures the structure of the system across three abstraction levels: Context, Container, and Component. This section provides a detailed description of the interactions with external systems, the internal software containers that comprise the framework, and the key components within each subsystem. The initial consideration of the implementation aims to facilitate GitOps-based workflows while providing actionable security analysis before containers are deployed.

A. Context-Level Architecture

The context-level view, illustrated in Fig.1, defines the boundary of the proposed Container Attestation Framework (CAF) and its interactions with external actors and systems.

A Software Engineer, acting either manually or via an automated pipeline, submits an attestation request for a container image. This request is processed by the CAF, which

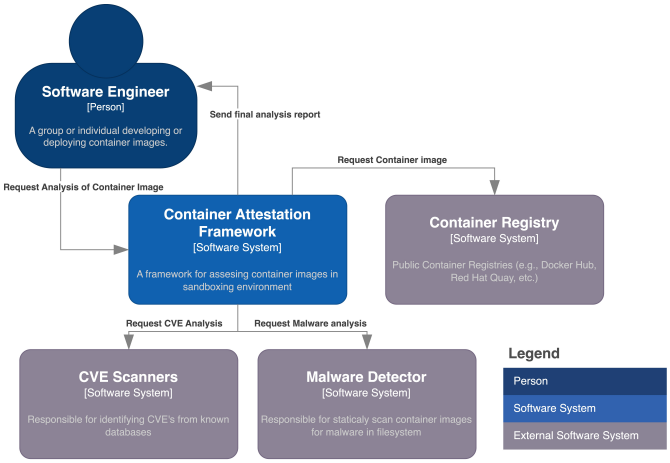

Fig. 1: Container Attestation Framework - Context Level

then performs static and dynamic analysis by orchestrating interactions with both external software systems and internal subsystems.

The framework retrieves the container image from a public container registry, such as Docker Hub, or RedHat Quay. Subsequently, two static analyses are triggered simultaneously:

1) A **CVE Scanner** analyzes the container's metadata and software packages to detect known vulnerabilities leveraging the National Vulnerability Database (NVD).
2) A **Malware Detector** receives the container image as a tarball and responds in a STIX-compliant threat report.

After collecting results from the static scans, the CAF executes the container sandboxed environment in our case, using gVisor. A secure user-space kernel that intercepts and emulates Linux syscalls. During this process, the framework captures and stores system call traces and network packets in a fully monitored environment without external connectivity.

Finally, the results from both static and runtime analyses are aggregated into a structured JSON report. Specifically, this report includes CVE summaries, the Malware detection results (STIX report), behavioral logs, and composite risk score. The final report is returned to the requesting user or system, depending on the type of trigger.

B. Container-Level Architecture

The container-level architecture, shown in Fig.2, decomposes the internal structure of the CAF into five loosely coupled containers, each responsible for a distinct phase of the attestation process.

1) **Container Processor**: This container is mostly used as an entry point into the system. It exposes a REST API for the attestation and dispatches tasks for execution to other components. Upon receiving an attestation request, it parses the container image name, downloads it locally, and generates a unique attestation identified (UUID). It triggers simultaneous requests for CVE analysis, malware detection, and sandbox execution tasks leveraging a distributed task queue.

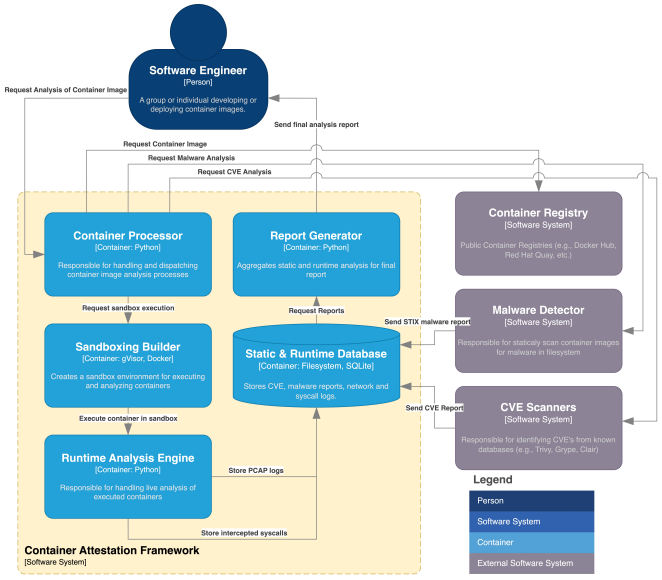

Fig. 2: Container Attestation Framework - Container Level

2) **Sandboxing Builder**: This container configures and initiates the sandbox environment using gVisor. Specifically, it registers runtime parameters, including volumes mounts for the logs generated during the runtime execution, it also sets the OCI runtime to runsc, it disables external networking, and sets debug/strace flags to capture system behavior. This configuration ensures full isolation and observability without compromising the host.
3) **Runtime Analysis Engine**: During execution, this container monitors the live behaviour of the attested container. In detail, system call activity is captured through strace using gVisor's native syscall hooks. Similarly, the network traffic is captured in a PCAP file where the loopback is only available across the network stack. Once the execution completes, which is bounded by a timeout event specified in the configuration option of the attestation request, logs are parsed and stored for downstream analysis.
4) **Static & Runtime Database**: Runtime logs, PCAP files, and external scan results are stored in two subsystem:
   - A SQLite database for structured reports (CVE and Malware metadata in STIX report)
   - A Filesystem directory for raw execution logs (e.g., PCAP, syscall trace files)
5) **Report Generator**: This container aggregates all data from the previous phases. It then computes a final risk score based on CVE severity, malware classification, and behavioral anomalies that have been observed at runtime. The report is then formatted into a JSON output, which includes the actual STIX report for completeness.

C. Component-Level Architecture

Following a more detailed break down of each container the component-level architecture is used. Fig.3 showcases the functional software components and clarifies how these modules coordinate to execute a container attestation pipeline, from request to report delivery.

The initiation is triggered as previously mentioned by a Software Engineer manually or automatically via a pipeline. The attestation request is received by the API handler, a lightweight REST endpoint built with the Python Flask Web Framework. The incoming request includes the image name, the image tag, and the repository source (e.g., Docker Hub) to which the docker client must connect to download. Upon receiving the attestation request, a unique attestation ID is attached for traceability across the attestation pipeline.

In order to manage the asynchronous task execution, a Task Queue implemented using Python Celery sits in the middle and dispatches three parallel jobs:

1) A CVE scan task
2) A malware detection task
3) A runtime sandbox execution task

Each task is handled by a designated Python client within the Container Processor container. The CVE Scan task is received by the CVE Analysis Client, which is responsible for communicating with the external CVE scanning services (such as Trivy, Clair or Gripe) via HTTP requests. After the scanning process a webhook for storing the resulting CVE data is used to persist them into the Security Analysis Database. Similarly, the malware detection task is received and handled by the Malware Scan Client. Responsible for converting the subject container into a tarball file and dispatching an HTTP request to the external Malware Detector service, in our case, the COSOCO malware detector. The response with the classification of the container is wrapped in a STIX threat report and saved into the Security Analysis Database.

Simultaneously, the Sandbox Manager receives the runtime sandbox execution task and prepares the environment for executing the container in a controlled and fully monitored runtime. Specifically, it configures the Docker daemon to use the runsc OCI runtime provided by gVisor, enabling syscall interception and isolation from the host kernel. Meanwhile, the Network access is disabled to further secure the sandbox, leaving only the loopback of the network stack. The strace and debug logging is enabled to capture the runtime behavior of the container. All generated logs are named using the unique attestation identifier generated by the API Handler in the Container Processor container.

Once the sandbox is ready to be used, the Sandboxing Builder takes over the execution. Given that the Sandboxing Orchestrator and the Sandbox Executor utilize the underlying docker daemon and gVisor, but their operations are different, they are illustrated separately in the figure. The Sandbox Orchestrator builds the execution environment by specifying the directories where the logs will be persisted. In contrast, the Sandbox Executor launches the container for time-constrained execution (e.g., 60 seconds) using the provided configurations.

During the execution of the container, the Runtime Analysis Engine in the background is activated with the gVisor runtime hooks configured by the Sandboxing Builder. These hooks

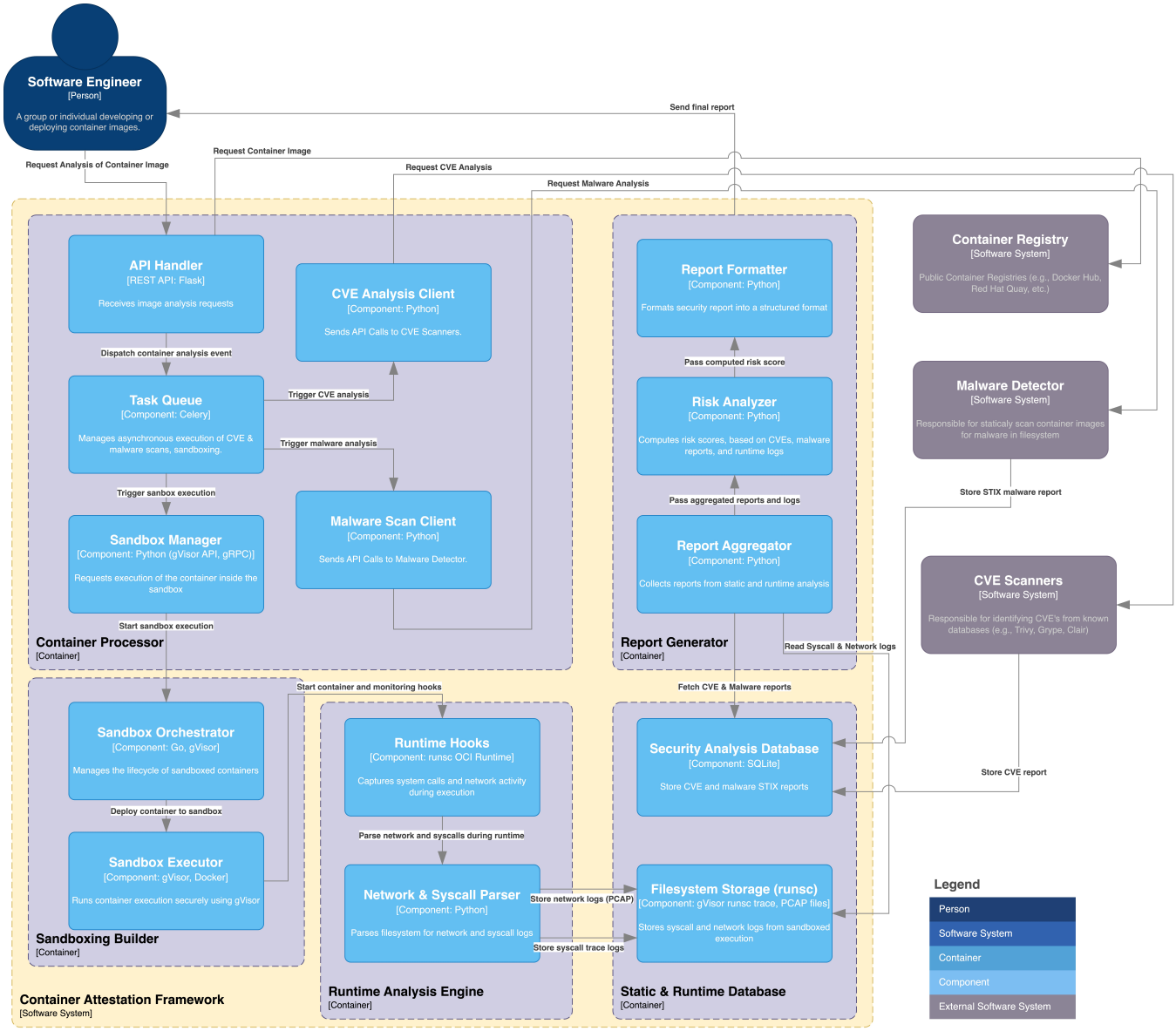

Fig. 3: Container Attestation Framework - Component Level

capture (i) all system calls made by the containerized process (via strace-compatible output) and (ii) internal network communication that is captured in PCAP format via the loopback interface.

After the duration for executing the container in the sandbox, the Network & Syscall Parser preprocesses the raw syscall logs and network captures to follow the naming convention of the unique attestation identifier. Afterwards, it then moves them into the Static & Runtime Database. Considering the type of the generated files, a remote or local object file storage can be leveraged instead.

As a final step, the Report Generator takes over. The Report Aggregator, using the unique attestation identifier, fetched the CVE results, the Malware STIX report, the runtime syscall logs and the network PCAP file. Once all are in place, the Risk Analyzer starts the assessment of the overall security score of the subject container. It correlates multiple sources of input such as CVE scores, Malware classification, anomalous runtime behaviors and network traces. Due to the requirement of domain-specific expertise and the broader scope such integration entails, this component is intentionally abstracted as a high-level placeholder in the current design. Potential integrations may include third-party analysis tools, large-language model (LLM) interfaces, intrusion detection systems, threat intelligence feeds, or statistical anomaly detectors. Finally, the Report Formatter serialises the output of the Risk Analyzer into a JSON response, which can be consumed by either a Software Engineer or an automated downstream process.

V. Conclusion & Future Work

This paper introduced a GitOps-compatible container attestation framework designed to enhance supply chain security by integrating static CVE analysis with dynamic sandboxing. The proposed architecture adopts a multi-layered approach that combines industry-standard vulnerability scanners with behavioral runtime monitoring via gVisor, producing a comprehensive security report in STIX format. By observing both the container's filesystem and its runtime behavior—including system calls and internal network traffic—the framework aims to identify threats that static analysis alone would miss, such as zero-day malware, hidden executables, or misconfigured containers.

Unlike existing solutions that focus on signature-based detection or hardened base images, our approach proactively evaluates existing images found in public container registries, regardless of their provenance or documentation (e.g., SBOM or SLSA metadata). This expands its applicability to legacy systems, community-contributed images, and third-party software commonly integrated into DevSecOps pipelines.

While the architecture and component breakdown offer a practical blueprint for implementation, several avenues exist for future work. First, integrating automated decision-making policies—such as blocking or flagging attested containers. Second, extending the framework with live threat intelligence feeds or anomaly detection models may further improve behavioral detection accuracy. Finally, deploying the framework in production-like Kubernetes environments (e.g., ArgoCD pipelines) will allow for GitOps-native container verification workflows at scale.

As modern software delivery accelerates, ensuring trust in containerised workloads becomes increasingly critical. This framework offers a foundational step toward that goal by enabling proactive, runtime-aware security attestation of container images.

References

[1] D. Reis, B. Piedade, F. F. Correia, J. P. Dias, and A. Aguiar, "Developing docker and docker-compose specifications: A developers' survey," *IEEE Access*, vol. 10, pp. 2318–2329, 2022.

[2] D. D. Murugiah Souppaya, Karen Scarfone, "Secure software development framework (ssdf)," https://doi.org/10.6028/NIST.SP.800-218, February 2022, [Accessed 11-04-2025].

[3] International Organization for Standardization, "ISO/IEC 27001:2022 - information security, cybersecurity and privacy protection — information security management systems — requirements," https://www.iso.org/standard/82875.html, 2022, accessed: 2025-04-11.

[4] SentinelOne, "Top 10 container security issues," https://www.sentinelone.com/cybersecurity-101/cloud-security/container-security-issues/#:~:text=As%20the%20threat%20landscape%20constantly,to%20mitigate%20container%20security%20issues, 2024, [Online; accessed 10-March-2025].

[5] Aquase, "Threat alert: Supply chain attacks using container images," https://www.aquasec.com/blog/supply-chain-threats-using-container-images/#:~:text=with%20a%20high%20risk%20of,Enact%20policies, 2021, [Online; accessed 10-March-2025].

[6] M. S. Haq, A. Ş. Tosun, and T. Korkmaz, "Security analysis of docker containers for arm architecture," in *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*. IEEE, 2022, pp. 224–236.

[7] M. Mounesan, H. Siadati, and S. Jafarikhah, "Exploring the threat of software supply chain attacks on containerized applications," in *2023 16th International Conference on Security of Information and Networks (SIN)*, 10 2023.

[8] iterasec, "Top container security vulnerabilities and how to solve them," https://iterasec.com/blog/container-security-vulnerabilities/#:~:text=, 2024, [Online; accessed 10-March-2025].

[9] aquasecurity, "Trivy a versatile security scanner," https://github.com/aquasecurity/trivy, [Online; accessed 10-March-2025].

[10] anchore, "A vulnerability scanner for container images and filesystems," https://github.com/anchore/grype, [Online; accessed 10-March-2025].

[11] quay, "Clair static analysis of vulnerabilities in application containers," https://github.com/quay/clair, [Online; accessed 10-March-2025].

[12] Amit Bismut, Idan Revivo, "Dynamic threat analysis for container images: Uncovering hidden risks," https://www.aquasec.com/blog/dynamic-container-analysis/#:~:text=This%20risk%20isn%E2%80%99t%20theoretical,%E2%80%93%20often%20using%20several%20stages, 2020, [Online; accessed 12-March-2025].

[13] Gal Singer, "Threat alert: Kinsing malware attacks targeting container environments," https://www.aquasec.com/blog/threat-alert-kinsing-malware-container-vulnerability/, 2020, [Online; accessed 12-March-2025].

[14] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling mec services in fast and secure way," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019, pp. 209–214.

[15] M. Jain, "Study of firecracker microvm," *CoRR*, vol. abs/2005.12821, 2020. [Online]. Available: https://arxiv.org/abs/2005.12821

[16] V. Vouvoutsis, F. Casino, and C. Patsakis, "Beyond the sandbox: Leveraging symbolic execution for evasive malware classification," *Computers & Security*, vol. 149, p. 104193, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740482400498X

[17] A. Nousias, E. Katsaros, E. Syrmos, P. Radoglou-Grammatikis, T. Lagkas, V. Argyriou, I. Moscholios, E. Markakis, S. Goudos, and P. Sarigiannidis, "Malware detection in docker containers: An image is worth a thousand logs," 2025. [Online]. Available: https://arxiv.org/abs/2504.03238

[18] "NVD - cve-2019-5736 — nvd.nist.gov," https://nvd.nist.gov/vuln/detail/cve-2019-5736, [Accessed 12-03-2025].

[19] M. Mounesan, H. Siadati, and S. Jafarikhah, "Exploring the threat of software supply chain attacks on containerized applications," in *2023 16th International Conference on Security of Information and Networks (SIN)*, 2023, pp. 1–8.